Cuadernos Didácticos



Análisis Y diseño Orientado a Objetos

Cuaderno Nº 1

Juan Manuel Cueva Lovelle
Departamento de Informática
Universidad de Oviedo

Oviedo, Abril 2003

Ingeniería Informática



Cuadernos Didácticos
Ingeniería Informática

Cuaderno Nº 1

Análisis y Diseño Orientado a Objetos

Autor:

J.M. Cueva Lovelle
Universidad de Oviedo - España

Editorial:

SERVITEC

ISBN: 84-8497-802-8
Deposito Legal: AS-783-98

1ª Edición: Oviedo, Marzo 1998
2º Edición: Oviedo, Abril 2000
3ª Edición: Oviedo, Febrero 2002

4ª Edición : Oviedo, Abril 2003

Consultor Editorial
Juan Manuel Cueva Lovelle
cueva@lsi.uniovi.es

Prólogo

El contenido de este Cuaderno Didáctico es un conjunto de transparencias que sirven para dar soporte a un primer curso de Análisis y Diseño Orientado a Objetos. Es por tanto un material de trabajo para facilitar el seguimiento de las explicaciones llevadas a cabo durante el desarrollo del curso. Este material debe ser completado con las notas de la persona que sigue el curso, el desarrollo de ejercicios y la lectura de las obras a las que se hace referencia al final de cada capítulo.

El objetivo de este Cuaderno es transmitir al alumno los conocimientos necesarios para afrontar el desarrollo de aplicaciones a gran escala, haciéndose especial hincapié en las partes de análisis y diseño orientado a objetos.

El desarrollo del curso se organiza en ocho temas. En los dos primeros temas se pretende motivar y justificar el uso de una metodología para desarrollar software.A continuación se desarrollan dos temas específicos de Tecnología de Objetos. Una vez que se conocen las motivaciones y las bases técnicas de la Tecnología de Objetos se afrontan los temas específicos de Análisis y Diseño orientado a objetos.

En el *tema 1: La complejidad del desarrollo de software*, se hace hincapié en el gran número de problemas que surgen cuando se quiere desarrollar software para sistemas complejos por parte de un equipo de personas.

Mientras que en el tema uno se presenta una visión pesimista del problema de desarrollar software, en el *tema 2: Calidad del software*, se pretende dar una visión más optimista indicándose como el seguimiento de una metodología contribuye obtener software de calidad.

En el *tema 3: Introducción a la Tecnología de Objetos*, se introducen los conceptos básicos necesarios para comprender la tecnología de objetos.

En el *tema 4: Clases y Objetos*, se desarrollan pequeños ejemplos concretos del uso de clases y objetos. También se marcan unas pautas para la construcción de clases.

En el *tema 5: Análisis Orientado a Objetos*, se presentan todas las fases del ciclo de desarrollo de software, así como los distintos pasos de la metodología que se aplicará. Se indicarán los principales modelos de análisis y definición de requisitos.

En el *tema 6: Diseño Orientado a Objetos* se presentan los distintos diagramas que se utilizan en la fase de diseño.

En el *tema 7: El proceso de desarrollo* se muestra una panorámica general de los propósitos de cada paso de la metodología detallándose los productos que se obtienen, las actividades que se realizan, los hitos que se alcanzan y las formas de medir la bondad de los productos obtenidos.

En el *tema 8: Aspectos prácticos* se plantean los problemas principales que ocurren en la gestión y planificación de grandes proyectos software.

El autor

Análisis y Diseño Orientado a Objetos
Contenidos

Tema 1: La complejidad del desarrollo de software
Tema 2: Calidad del software
Tema 3: Introducción a la Tecnología de Objetos
Tema 4: Clases y objetos
Tema 5: Análisis Orientado a Objetos
Tema 6: Diseño Orientado a Objetos
Tema 7: El proceso de desarrollo
Tema 8: Aspectos prácticos

Bibliografía básica

• G. Booch, J. Rumbaugh, I. Jacobson. *El lenguaje unificado de modelado.* Addison-Wesley (1999)
• J. Rumbaugh, I. Jacobson, G. Booch, . *El lenguaje unificado de modelado. Manual de referencia.* Addison-Wesley (1999)
• I. Jacobson G. Booch, J. Rumbaugh. *El proceso Unificado de Desarrollo de Software.* Addison-Wesley (1999)
• G.Booch. *Análisis y diseño orientado a objetos con aplicaciones.* 2ª Edición. Addison-Wesley/ Díaz de Santos (1996).
• B. Meyer. *Construcción de software orientado a objetos. Segunda edición*. Prentice-Hall (1999).
• E. Gamma, R. Helm, R. Johnson, J Vlissides. *Patrones de diseño. Elementos de software orientado a objetos reutilizable*. Addisson-Wesley/ Pearson Educación (2003).

Bibliografía básica para C++

• L. Joyanes Aguilar. *Programación en C++. Algoritmos, Estructuras de datos y objetos.* McGrawHill (1999).
• B. Stroustrup. *El lenguaje de programación C++*. Edición especial. Addison-Wesley (2002).
• J. P. Cohoon, J. W. Davidson. *Programación y diseño en C++.* Segunda edición. McGraw-Hill (2000)

Bibliografía básica para Java

• P. Niemeyer, J. Knudsen. *Curso de Java. ANAYA/O'Reilly* (2000).
• K. Arnold, J. Gosling. *El lenguaje de programación Java. Addison-Wesley (1997)*
• L. Joyanes, M Fernández Azuela. *Java2:Manual de programación.* McGraw-Hill (2002)

Bibliografía básica para C# y .NET

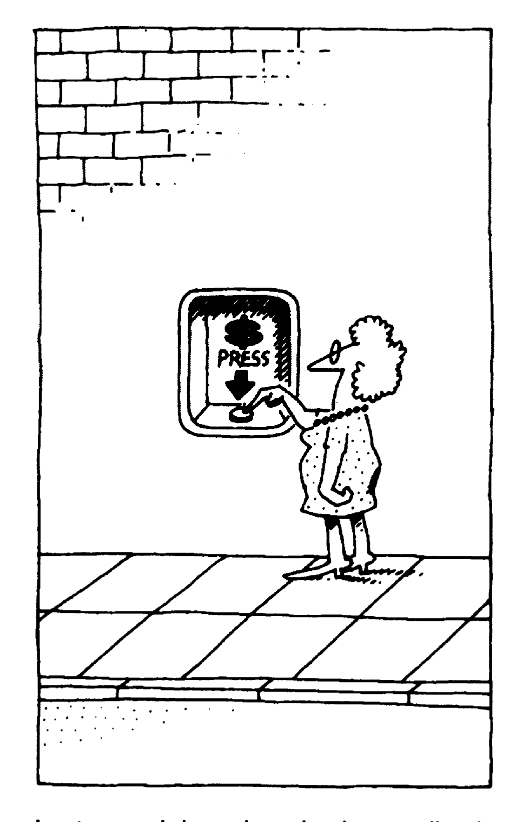
• D. S. Platt. *Así es Microsoft .NET*. McGraw-Hill (2001)
• L. Joyanes, M Fernández Azuela. *C#:Manual de programación.* McGraw-Hill (2002)

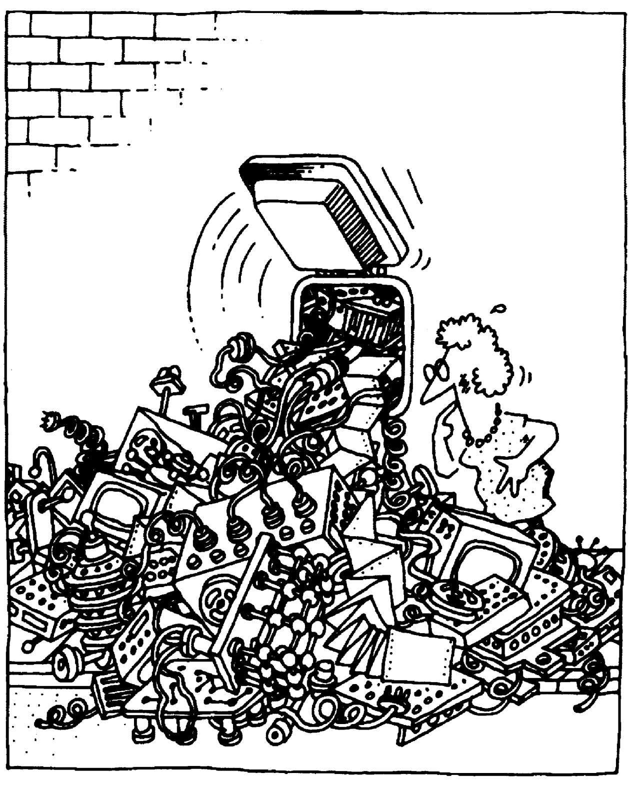
Bibliografía complementaria

• L. Joyanes Aguilar. *Programación Orientada a Objetos. Segunda Edición.* McGrawHill (1998).
• Pierre-Alain Muller. *Modelado de objetos con UML.* Eyrolles-Gestión 2000 (1997).
• N. López, J. Migueis, E. Pichon. *Integrar UML en los proyectos.* Eyrolles-Gestión 2000 (1998).
• R.C. Martin. *Designing Object-Oriented C++ applications using the Booch method* Prentice-Hall (1995)
•J.M. Cueva et al. *Introducción a la programación estructurada y orientada a objetos con Pascal.* Distribuido por Editorial Ciencia-3 (1994).

Tema 1º
La complejidad del desarrollo de software

Contenidos





La tarea del equipo de desarrollo de software es ofrecer ilusión de simplicidad.

[Booch 94]

1.1 La complejidad inherente al software

La construcción de software puede involucrar elementos de gran complejidad, que en muchos casos no son tan evidentes como los que se pueden ver en otras ingenierías. Un puente, un edificio, una mina, una red de ferrocarriles son ejemplos de sistemas complejos de otras ingenierías, pero el ingeniero del software construye sistemas cuya complejidad puede parecer que permanece oculta. El usuario siempre supone que Informática todo es muy fácil ("apretar un botón y ya está")

La complejidad se estudiará en los siguientes apartados: [Booch 94]

- Las propiedades de los sistemas de software simples y complejos
- ¿Por qué el software es complejo de forma innata?
- Las consecuencias de la complejidad ilimitada

Propiedades de los sistemas de software simples o artesanales

- No son complejos
- Suelen estar construidos y mantenidos por una sola persona (software artesanal)
  - No suelen pasar de 1.000.000 de líneas de código, aunque el número de líneas de código no sea la mejor medida de la complejidad del software.
- Ciclo de vida corto
  - El ciclo de vida del software es todo el tiempo que un proyecto software está activo, comienza con las primeras tareas de análisis y no concluye hasta que ese software deja de utilizarse por los usuarios
- Pueden construirse aplicaciones alternativas en un periodo razonable de tiempo
- No necesitan grandes esfuerzos en análisis y diseño
- No son nuestro objetivo de estudio

Las propiedades de los sistemas software complejos

- También se denominará software de dimensión industrial
- Es muy difícil o imposible que un desarrollador individual pueda comprender todas las sutilidades de su diseño
- La complejidad es una propiedad esencial de estos sistemas, que puede dominarse, pero no eliminarse
- Son el objetivo de nuestro estudio
- Ejemplos: Un sistema de reservas, anulaciones y ventas de billetes aéreos para un conjunto de compañías aéreas que se pueda utilizar en cualquier lugar del mundo.

¿Por qué el software es complejo de forma innata?

La complejidad inherente al software se deriva de los siguientes cuatro elementos [Booch 94]

- La complejidad del dominio del problema
- La dificultad de gestionar el proceso de desarrollo
- El detalle que se puede alcanzar a través del software
- El problema de caracterizar el comportamiento de sistemas discretos

La complejidad del dominio del problema

- **Gran cantidad de requisitos que compiten entre sí, incluso contradiciéndose**
  - La forma habitual de especificar los requisitos son grandes cantidades de texto con unos pocos dibujos
- **Desacoplamientos de impedancias entre usuarios del sistema y desarrolladores**
  - Los usuarios suelen tener ideas vagas de lo que desean
  - Dificultades de comunicación
  - Distintas perspectivas de la naturaleza del problema
- **Modificación de los requisitos con el paso del tiempo, pues los usuarios y desarrolladores comienzan a compenetrarse mejor**

  "No sobrevivirán las especies más fuertes, tampoco las más inteligentes, lo harán aquellas que más se adapten al cambio" ***Charles Darwin***, El origen de las especies.
  - Mantenimiento de software
    - Cuando se corrigen errores
  - Evolución del software
    - Cuando se responde a requisitos que cambian
  - Conservación del software
    - Se emplean medios extraordinarios para mantener en operación un elemento software anticuado y decadente

La dificultad de gestionar el proceso de desarrollo

- Dirigir un equipo de desarrolladores
  - mantener la unidad de acción
  - conservar la integridad del diseño
- Manejar gran cantidad de código
  - Uso de herramientas
    - gestión de proyectos
    - Análisis, diseño e implementación
    - Desarrollo rápido de prototipos (RAD)
    - Control de versiones
  - Uso de lenguajes de muy alto nivel
  - Reutilización de código
  - Uso de frameworks (marcos estructurales)
  - Uso de componentes software
  - Uso de patrones de diseño

El detalle que se puede alcanzar a través del software

"No vuelvas a inventar la rueda"

- Evitar el desarrollo de todo desde el nivel más inferior
- Utilizar los estándares, al igual que en otras ingenierías
- Verificar la fiabilidad de los estándares existentes en el mercado

El problema de caracterizar el comportamiento de sistemas discretos

- La ejecución de un programa es una transición entre estados de un sistema discreto
- Cada estado contiene las variables, su valor, direcciones en tiempo de ejecución, etc.
- Diseñar el sistema de forma que el comportamiento de una parte del sistema tenga mínimo impacto en otra parte del mismo
- La transición entre estados debe ser determinista
  – Sin embargo a veces no lo es, pues un evento puede corromper el sistema pues sus diseñadores no lo tuvieron en cuenta
- Es imposible hacer una prueba exhaustiva
  – Es decir probar todos los casos posibles, pues aunque el número de estados es finito el determinar todas las combinaciones posibles produce un número tan elevado de combinaciones que es imposible de probar cada una de ellas

Las consecuencias de la complejidad ilimitada

"Cuanto más complejo es un sistema más probable es que se caiga"

- La crisis del software
  - *Son los sucesivos fracasos de las distintas metodologías para dominar la complejidad del software, lo que implica el retraso de los proyectos de software, las desviaciones por exceso de los presupuestos fijados y la existencia de deficiencias respecto a los requisitos del cliente* [Booch 94]
  - Este término alude al conjunto de problemas que aparecen en el desarrollo de software [Pressman 97, apartado 1.3]
  - Muchas de las causas de la crisis del software son los mitos del sofware [Pressman 97, apartado 1.4]
    - Mitos de gestión
      - Si fallamos en la planificación, podemos añadir más programadores y adelantar el tiempo perdido
    - Mitos del cliente
      - Una declaración general de los objetivos es suficiente para comenzar a escribir programas; podemos dar los detalles más adelante
    - Mitos de los desarrolladores
      - Una vez que escribimos el programa y hacemos que funcione nuestro trabajo ha terminado
      - Hasta que no se ejecuta el programa no hay forma de comprobar su calidad
      - Lo único que se entrega al terminar el proyecto es el programa funcionando

Fallo del primer vuelo del Ariane 5

"El fallo del vuelo 501 se debió a la pérdida completa de la información de guiado y altitud, 37 segundos después de la ignición del motor principal (30 segundos después del despegue). Esta pérdida de información tiene su origen en ***errores de diseño y especificación del software*** *del Sistema de Referencia Inercial"*

Comisión de Investigación del fallo del primer vuelo del Ariane 5

TRIBUNA DE ASTRONOMÍA, nº 130, pp. 81, 1996.

Véase la página web [ARIAN501]

Este fallo ha dado lugar a varios artículos [Meyer 97].



1.2 La estructura de los sistemas complejos

"¿Cómo puede cambiarse esta imagen desoladora? Ya que el problema subyacente surge de la complejidad inherente al software, la sugerencia que se plantea aquí es estudiar cómo se organizan los sistemas complejos en otras disciplinas. Realmente, si se hecha una mirada al mundo que nos rodea, se observarán sistemas con éxito dentro de una complejidad que habrá que tener en cuenta." [Booch 94, capítulo 1]

- Ejemplos de sistemas complejos
- Los cinco atributos de un sistema complejo
- Complejidad organizada y desorganizada

Ejemplos de sistemas complejos

- La estructura de un ordenador personal
  - Su complejidad se domina por medio de una estructura jerárquica basada en componentes
  - Puede razonarse como funciona el ordenador debido a que puede descomponerse en partes susceptibles de ser estudiadas por separado.
  - Los niveles de la jerarquía son niveles de abstracción
  - Para resolver cada problema se elige un determinado nivel de abstracción
- La estructura de plantas y animales
  - Estructura jerárquica (célula, tejido,…)
  - Agentes (savia, sangre,…) tienen comportamiento complejo y contribuye a funciones de nivel superior.
- La estructura de la materia
  - Estructura jerárquica (átomos, electrones, protones, neutrones, quarks)
- La estructura de las instituciones sociales
  - Estructuras jerárquicas que evolucionan con el tiempo

Los cinco atributos de un sistema complejo

- **Jerarquía**
  - Un sistema complejo se compone de subsistemas relacionados que tienen a su vez sus propios subsistemas, y así sucesivamente hasta que se alcanza algún nivel ´´ínfimo de componentes elementales.
- **Componentes**
  - La elección de qué componentes de un sistema son primitivos depende de la decisión del analista.
- **Enlaces entre componentes**
  - Los enlaces internos de los componentes son más fuertes que los enlaces externos entre los componentes.
- **Patrones**
  - Los sistemas complejos tienen patrones comunes, que permiten la reutilización de componentes. Así los sistemas jerárquicos están compuestos usualmente de sólo unas pocas clases diferentes de subsistemas en varias combinaciones y disposiciones que se pueden encontrar en sistemas aparentemente muy diferentes.
- **Evolución**
  - Los sistemas complejos que funcionan son la evolución de sistemas simples que funcionaron previamente como formas intermedias estables.

Complejidad organizada y desorganizada

- **La forma canónica de un sistema complejo**
  - El descubrimiento de abstraciones y mecanismos comunes facilita en gran medida la comprensión de sistemas complejos
  - Dos tipos de jerarquías
    - Jerarquía estructural (“parte-de”, ”part of “, o de clases)
    - Jerarquía de tipos (“es un”, “is a”, o de objetos)
  - Combinando los dos tipos de jerarquías con los cinco atributos de un sistema complejo se pueden definir todos los sistemas complejos de forma canónica (la forma menos compleja)
  - Se denominará arquitectura de un sistema a la estructura de clases y objetos
- **Las limitaciones de la capacidad humana para enfrentarse a la complejidad**
  - La complejidad desorganizada es la que se nos presenta antes de determinar las abstracciones y jerarquías.
  - En los sistemas discretos hay que enfrentarse con un espacio de estados claramente grande, intrincado y a veces no determinista
  - Una persona no puede dominar todos los detalles de un sistema complejo
  - La complejidad se organizará según el modelo de objetos

1.3 Imponiendo orden al caos

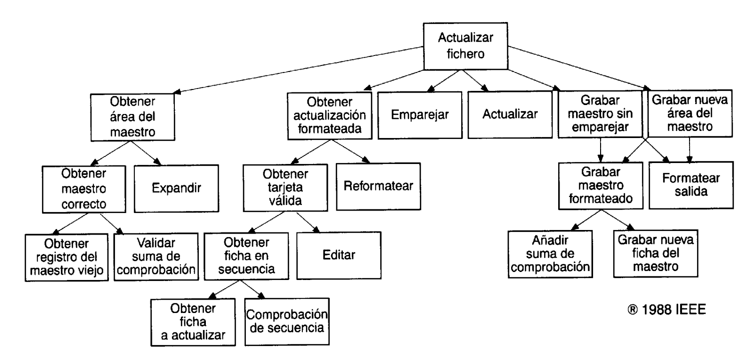
Complejidad organizada

- El papel de la descomposición
- El papel de la abstracción
- El papel de la jerarquía

El papel de la descomposición

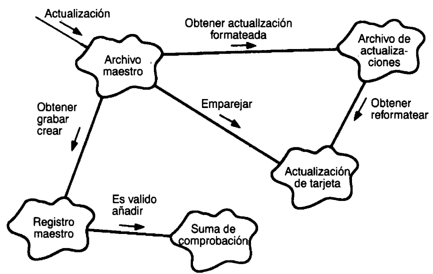
"La técnica de dominar la complejidad se conoce desde tiempos remotos: *divide et impera* (divide y vencerás)"

- **Descomposición algorítmica**
  - Por ejemplo por diseño estructurado descendente [Booch 94]



Descomposición algorítmica.

- **Descomposición orientada a objetos**
  - Se identifican objetos que hacen cosas cuando reciben mensajes [Booch 94]



Descomposición orientada a objetos.

- **Descomposición algoritmica versus descomposición orientada a objetos**
  - Visión algorítmica resalta el orden de los eventos
  - Visión orientada a objetos enfatiza los agentes que causan o padecen acciones
  - Ambas visiones son impotantes y ortogonales

Ventajas de la descomposición orientada a objetos sobre la descomposición algorítmica

- Los sistemas son más pequeños y fáciles de mantener a través de mecanismos de reutilización
- Los sistemas soportan mejor los cambios, pues están mejor preparados para evolucionar en el tiempo, dado que su diseño se basa en formas intermedias estables
- Se refleja mejor el comportamiento de los distintos elementos del sistema complejo al usar objetos
- Se reducen riesgos en los sistemas complejos al acercarse el mundo real compuesto por objetos al mundo del software
- La descomposición algorítmica no contempla los problemas de abstracción de datos y ocultación de información, ni proporciona medios adecuados para tratar la concurrencia

El papel de la abstracción

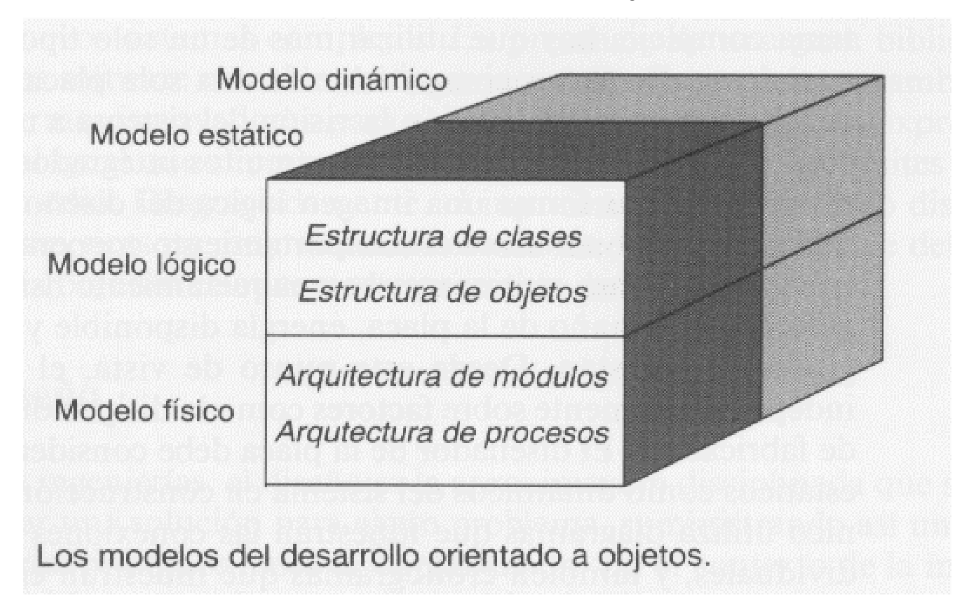
- La abstracción permite ignorar los detalles no esenciales de un objeto complejo
- La abstracción permite modelos generalizados e idealizados de objetos
- La abstracción reduce la complejidad
- Los objetos son abstracciones del mundo real, que agrupan información densa y cohesiva

El papel de la jerarquía

- El reconocimiento explícito de las jerarquías de clases y objetos dentro de un sistema software complejo es una forma de incrementar el contenido semántico de la información del sistema
- La jerarquía de objetos ilustra como diferentes objetos colaboran entre sí a través de patrones de iteracción denominados mecanismos
- La jerarquía de clases resalta la estructura y los comportamientos comunes en el interior del sistema
- La identificación de las jerarquías no es una tarea fácil
- La jerarquía de clases y objetos incluye herencia y composición.

1.4 Diseño de sistemas complejos

- **La ingeniería como ciencia y arte**
  - La práctica de cualquier disciplina de ingeniería (sea civil, mecánica o informática involucra elementos tanto de ciencia como de arte
  - La concepción de un diseño de un sistema software suele involucrar un salto de imaginación, una síntesis de experiencia y conocimiento como la que requiere cualquier artista
  - Una vez que el ingeniero ha articulado un diseño como artista, debe analizarlo técnicamente aplicando la sistemática de una metodología.
- **El significado del diseño**
  - El diseño es la aproximación disciplinada que se utiliza para concebir una solución para algún problema, suministrando así un camino desde los requisitos hasta la implementación.
  - Los productos del diseño son modelos que permiten razonar sobre las estructuras, hacer concesiones cuando los requisitos entran en conflicto y proporcionar un anteproyecto para la implementación
    - **La importancia de construir un modelo**
      - Un modelo es una representación simplificada de la realidad, que permite simular bajo situaciones tanto previstas como improbables el comportamiento del sistema
    - **Los elementos de los métodos de diseño de software**
      - **Notación**: El lenguaje para expresar cada modelo
      - **Proceso**: Actividades que se siguen para la construcción ordenada de los modelos del sistema
      - **Herramientas:** Los artefactos que facilitan la construcción del modelo, eliminando las tareas tediosas y comprobando errores e inconsistencias
    - **Los modelos del desarrollo orientado a objetos**



Los modelos del desarrollo orientado a objetos.

1.5 Resumen

- Las técnicas de análisis y diseño orientado a objetos están pensadas para desarrollo de software de dimensión industrial.
- El software es complejo de forma innata. La complejidad de los sistemas excede frecuentemente la capacidad intelectual humana
- La complejidad puede atacarse por medio del uso de la descomposición, abstracción y jerarquía (complejidad organizada)
- Los sistemas complejos pueden evolucionar desde formas intermedias estables
- El diseño orientado a objetos define una descomposición, abstracción y jerarquía basada en clases y objetos
- El diseño orientado a objetos define notaciones y procesos que conducen a modelos que permiten razonar sobre diferentes aspectos del sistema que se está simulando.
- Conviene volver a leer este tema cuando se tenga más experiencia en Análisis y Diseño

1.6 Test de Autoevaluación

1.6.1 Las técnicas de análisis y diseño orientadas a objetos **A)** Sólo se aplican para desarrollo de software artesanal o individual **B)** Están pensadas para desarrollo de software de dimensión industrial **C)** Sólo se aplican cuando se utilizan bases de datos **D)** Sólo se aplican en sistemas de tiempo real **E)** Ninguna afirmación es correcta.

1.6.2 La crisis del software es **A)** La perdida del monopolio de IBM en el campo del software **B)** El dominio de Microsoft en el mercado mundial de software **C)** Son los sucesivos fracasos de las distintas metodologías para dominar la complejidad del software **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta.

1.6.3 Si se nos pide el análisis y diseño de un producto de software que ha sido creado y mantenido por una sola persona en un periodo corto de tiempo. Se puede decir que **A)** Es un sistema de software simple **B)** No necesita grandes esfuerzos de análisis y diseño **C)** Se pueden construir aplicaciones alternativas en un periodo razonable de tiempo **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta

1.6.4 Si se nos pide construir un software determinado, el seguimiento de una metodología **A)** Facilita la producción de software de calidad **B)** Una forma de dominar la complejidad del software **C)** Es una colección de métodos aplicados a lo largo del ciclo de vida del desarrollo del software y unificados por alguna aproximación general o filosófica **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta

Respuestas: 1.6.1 B), 1.6.2 C), 1.6.3 D), 1.6.4 D)

1.7 Lecturas recomendadas

La mayor parte de los conceptos de este tema están explicados en el capítulo 1 del libro [Booch 94]. También se recomienda la lectura de la sección I (The Problem), capítulos 1 a 9 del libro [Beck 2000].

1.8 Referencias

- [ARIAN501] *Rapport de la Comission d'enquête Ariane 501.* `http://www.cnes.fr/actualites/news/rapport_501.html`
- [Beck 2000] K. Beck. *Extreme Programming explained. Embrace change.* Addison-Wesley, 2000.
- [Booch 94] G.Booch. *Object-Oriented Analysis and Design with Applications. Second Edition*. Addison-Wesley (1994). Versión Castellana: *Análisis y diseño orientado a objetos con aplicaciones.* 2ª Edición. Addison-Wesley/ Díaz de Santos (1996).
- [Meyer 97] J.-M. Jézéquel, B. Meyer. *Design by Contract: the lessons of Ariane*. IEEE Computer. January 1997, Vol. 30, No. 1, pp 129-130.
- [Pressman 97] R.S.Pressman. *Software engineering: a practitioner´s approach*. McGraw-Hill 4 Ed. (1997). Versión castellana: *Ingeniería del software: Un enfoque práctico.* 4ª Edición. McGraw-Hill(1998).
- [Tribuna Astronomía 130, 1996] Comisión de Investigación del fallo del primer vuelo del Ariane 5. TRIBUNA DE ASTRONOMÍA, nº 130, pp.81, 1996

Tema 2º: Calidad del software









INTERNATIONAL
ORGANIZATION FOR
STANDARDIZATION





2.1 Calidad del software

- Todas las metodologías y herramientas tienen un único fin ***producir software de gran calidad***
- Definiciones de calidad del software
  - *"Concordancia con los requisitos funcionales y de rendimiento explícitamente establecidos con los estándares de desarrollo explícitamente documentados y con las características implícitas que se espera de todo software desarrollado profesionalmente"* [Pressman 93]
  - *"El conjunto de características de una entidad que le confieren su aptitud para satisfacer las necesidades expresadas y las implícitas"* **ISO 8402 (UNE 66-001-92)**.[ISO][AENOR]
- Conclusiones
  - Los *requisitos del software* son la base de las medidas de calidad. La falta de concordancia con los requisitos es una falta de calidad
  - Los estándares o *metodologías* definen un conjunto de criterios de desarrollo que guían la forma en que se aplica la ingeniería del software. Si no se sigue ninguna metodología siempre habrá falta de calidad
  - Existen algunos *requisitos implícitos* o *expectativas* que a menudo no se mencionan, o se mencionan de forma incompleta (por ejemplo el deseo de un buen mantenimiento) que también pueden implicar una falta de calidad.

2.2 Aseguramiento de calidad del software

(Software Quality Assurance)

- El aseguramiento de calidad del software *es el conjunto de actividades planificadas y sistemáticas necesarias para aportar la confianza en que el producto (software) satisfará los requisitos dados de calidad.*
- El aseguramiento de calidad del software se diseña para cada aplicación antes de comenzar a desarrollarla y no después.
- Algunos autores prefieren decir **garantía** de calidad en vez de aseguramiento.
  - Garantía, puede confundir con garantía de productos
  - Aseguramiento pretende dar confianza en que el producto tiene calidad
- El aseguramiento de calidad del software está presente en
  - Métodos y herramientas de análisis, diseño, programación y prueba
  - Inspecciones técnicas formales en todos los pasos del proceso de desarrollo del software
  - Estrategias de prueba multiescala
  - Control de la documentación del software y de los cambios realizados
  - Procedimientos para ajustarse a los estándares (y dejar claro cuando se está fuera de ellos)
  - Mecanismos de medida (métricas)
  - Registro de auditorias y realización de informes
- Actividades para el aseguramiento- de calidad del software
  - Métricas de software para el control del proyecto
  - Verificación y validación del software a lo largo del ciclo de vida
    - Incluye las pruebas y los procesos de revisión e inspección
  - La gestión de la configuración del software

2.3 Gestión de la calidad del software
(Software Quality Management)

- Gestión de la calidad (ISO 9000)
  - "*Conjunto de actividades de la función general de la dirección que determina la calidad, los objetivos y las responsabilidades y se implanta por medios tales como la planificación de la calidad, el control de la calidad, el aseguramiento (garantía) de la calidad y la mejora de la calidad, en el marco del sistema de calidad*"[ISO][AENOR]
- Política de calidad (ISO 9000)
  - "*Directrices y objetivos generales de una organización, relativos a la calidad, tal como se expresan formalmente por la alta dirección*"[ISO][AENOR]
- La gestión de la calidad se aplica normalmente a nivel de empresa
- También puede haber una gestión de calidad dentro de la gestión de cada proyecto

2.4 Control de la calidad del software

(*Software Quality Control*)

- *"Son las técnicas y actividades de carácter operativo, utilizadas para satisfacer los requisitos relativos a la calidad, centradas en dos objetivos fundamentales:*
  - *mantener bajo control un proceso*
  - *eliminar las causas de los defectos en las diferentes fases del ciclo de vida"* [ISO][AENOR]
- En general son las actividades para evaluar la calidad de los productos desarrollados

2.5 Sistema de calidad

- Sistema de calidad
  "Estructura organizativa, procedimientos, procesos y recursos necesarios para implantar la gestión de calidad"[ISO][AENOR]
- El sistema de calidad se debe adecuar a los objetivos de calidad de la empresa
- La dirección de la empresa es la responsable de fijar la política de calidad y las decisiones relativas a iniciar, desarrollar, implantar y actualizar el sistema de calidad.
- Un sistema de calidad consta de varias partes
  - Documentación
    - Manual de calidad. Es el documento principal para establecer e implantar un sistema de calidad. Puede haber manuales a nivel de empresa, departamento, producto, específicos (compras, proyectos,…)
  - Parte física: locales, herramientas ordenadores, etc.
  - Aspectos humanos:
    - Formación de personal
    - Creación y coordinación de equipos de trabajo
- Normativas
  - ISO [ISO][AENOR]
    - ISO 9000: Gestión y aseguramiento de calidad (conceptos y directrices generales)
    - Recomendaciones externas para aseguramiento de la calidad (ISO 9001, ISO 9002, ISO 9003)
    - Recomendaciones internas para aseguramiento de la calidad (ISO 9004)
  - MALCOM BALDRIGE NATIONAL QUALITY AWARD [NIST]
  - Software Engineering Institute (SEI) Capability Maturity Model (CMM) for software [SEI]
  - Administración pública española. *Plan General de Garantía de Calidad aplicable al desarrollo de equipos lógicos (PGGC)* [MAP].

2.6 Certificación de la calidad
(Quality certification)

- Un sistema de certificación de calidad permite una valoración independiente que debe demostrar que la organización es capaz de desarrollar productos y servicios de calidad
- Los pilares básicos de la certificación de calidad son tres [Sanders 1994, p. 44] :
  - Una metodología adecuada
  - Un medio de valoración de la metodología
  - La metodología utilizada y el medio de valoración de la metodología deben estar reconocidos ampliamente por la industria

2.7 Factores que determinan la calidad de un producto software

[McCall 1977] [Meyer 1997, capítulo 1] [Pressman 1998, capítulo 18]

Se clasifican en tres grupos:

- **Operaciones del producto**: características operativas
  - *Corrección* (¿Hace lo que se le pide?)
    - El grado en que una aplicación satisface sus especificaciones y consigue los objetivos encomendados por el cliente
  - *Fiabilidad* (¿Lo hace de forma fiable todo el tiempo?)
    - El grado que se puede esperar de una aplicación lleve a cabo las operaciones especificadas y con la precisión requerida
  - *Eficiencia* (¿Qué recursos hardware y software necesito?)
    - La cantidad de recursos hardware y software que necesita una aplicación para realizar las operaciones con los tiempos de respuesta adecuados
  - *Integridad* (¿Puedo controlar su uso?)
    - El grado con que puede controlarse el acceso al software o a los datos a personal no autorizado
  - *Facilidad de uso* (¿Es fácil y cómodo de manejar?)
    - El esfuerzo requerido para aprender el manejo de una aplicación, trabajar con ella, introducir datos y conseguir resultados
- **Revisión del producto**: capacidad para soportar cambios
  - *Facilidad de mantenimiento* (¿Puedo localizar los fallos?)
    - El esfuerzo requerido para localizar y reparar errores
  - *Flexibilidad* (¿Puedo añadir nuevas opciones?)
    - El esfuerzo requerido para modificar una aplicación en funcionamiento
  - *Facilidad de prueba* (¿Puedo probar todas las opciones?)
    - El esfuerzo requerido para probar una aplicación de forma que cumpla con lo especificado en los requisitos
- **Transición del producto**: adaptabilidad a nuevos entornos
  - *Portabilidad* (¿Podré usarlo en otra máquina?)
    - El esfuerzo requerido para transferir la aplicación a otro hardware o sistema operativo
  - *Reusabilidad* (¿Podré utilizar alguna parte del software en otra aplicación?)
    - Grado en que partes de una aplicación pueden utilizarse en otras aplicaciones
  - *Interoperabilidad* (¿Podrá comunicarse con otras aplicaciones o sistemas informáticos?
    - El esfuerzo necesario para comunicar la aplicación con otras aplicaciones o sistemas informáticos

2.8 Métricas de la calidad de un producto sofware

[Pressman 1998, capítulo 18]

- Es difícil, y en algunos casos imposible, desarrollar medidas directas de los factores de calidad del software
- Cada factor de calidad F_c se puede obtener como combinación de una o varias métricas:

$$F_c = c_1 * m_1 + c_2 * m_2 + \ldots + c_n * m_n$$

  - c_i factor de ponderación de la métrica i, que dependerá de cada aplicación específica
  - m_i métrica i
  - Habitualmente se puntúan de 0 a 10 en las métricas y en los factores de calidad
- Métricas para determinar los factores de calidad
  - Facilidad de auditoria
  - Exactitud
  - Normalización de las comunicaciones
  - Completitud
  - Concisión
  - Consistencia
  - Estandarización de los datos
  - Tolerancia de errores
  - Eficiencia de la ejecución
  - Facilidad de expansión
  - Generalidad
  - Independencia del hardware
  - Instrumentación
  - Modularidad
  - Facilidad de operación
  - Seguridad
  - Auto-documentación
  - Simplicidad
  - Independencia del sistema software
  - Facilidad de traza
  - Formación

2.9 Auditoria informática

[Piattini 2001]

"La auditoria informática es el proceso de recoger, agrupar y evaluar evidencias para determinar si un sistema informatizado salvaguarda los activos, mantiene la integridad de los datos, lleva a cabo eficazmente los fines de la organización y utiliza eficientemente los recursos"

Los objetivos fundamentales de la auditoria informática son

- Protección de activos e integridad de datos
- Eficacia y eficiencia en la gestión de recursos

Principales áreas de la auditoria informática

- Auditoria física
- Auditoria de la ofimática
- Auditoria de la dirección
- Auditoria de la explotación
- Auditoria del desarrollo
- Auditoria del mantenimiento
- Auditoria de bases de datos
- Auditoria de técnica de sistemas
- Auditoria de la calidad
- Auditoria de la seguridad
- Auditoria de redes
- Auditoria de aplicaciones
- Auditoria de los sistemas de soporte a la toma de decisiones
- Auditoria jurídica de entornos informáticos

El resultado de la auditoria informática es el Informe de Auditoria Informática

2.10 Test de auto-evaluación

2.10.1 Los pilares básicos de la certificación de calidad del software son **A)** Una metodología adecuada **B)** Un medio de valoración de la metodología **C)** Un reconocimiento de la industria de la metodología utilizada y del medio de valorar la metodología **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta.

2.10.2 La calidad del software implica **A)** La concordancia entre el software diseñado y los requisitos **B)** Seguir un estándar o metodología en el proceso de desarrollo de software **C)** Tener en cuenta los requisitos implícitos (no expresados por los usuarios) **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta

Respuestas: 2.10.1 D) 2.10.2 D)

2.11 Lecturas recomendadas

Como primera lectura sobre calidad del software se recomienda el capitulo 13 de [Piattini 1996] y el capítulo 8 de [Pressman 1998]. Para una visión sobre la situación actual y nuevas tendencias léase el número monográfico de la revista [NOVATICA 137].

Para consultar las diversas normativas de calidad se recomienda navegar por las páginas web del [ISO],[AENOR],[NIST],[SEI], [MAP].

Sobre auditoria se recomienda leer en una primera aproximación el libro [Piattini 2001]

2.12 Referencias

[AENOR] *AENOR Asociación Española de Normalización y Certificación.. http://www.aenor.es*

[ISO] *International Standardization Organization. http://www.iso.ch*

[Jackson 1996] P. Jackson, D. Ashton. *Implemente calidad de clase mundial. ISO 9000-BS5750.* Limusa (1996).

[Kan 1995] S. H. Kan. *Metrics and Models in software Quality Engineering*. Addison-Wesley (1995).

[MAP] Ministerio Administraciones Públicas de España. Consejo Superior de Informática. http://www.map.es/csi

[McCall 1977] James McCall (Editor).*Factors in Software Quality*. Technical Report, General Electric, 1977.

[Meyer 1997] B. Meyer. *Object-Oriented software construction*.Second Edition, Prentice-Hall, 1997. Versión castellana: *Construcción de software orientado a objetos,*Prentice-Hall, 1999.

[NIST] National Institute of Standars and Technology. http://www.quality.nist.gov/

Norma ISO 9000-1 UNE (31 páginas). International Standardization Organization. Una Norma Española. http://www.aenor.es

Norma ISO 9001 UNE (21 páginas) International Standardization Organization. Una Norma Española. http://www.aenor.es [ISO]

Norma ISO 9000-3 (5 + 15 páginas) International Standardization Organization. Una Norma Española. http://www.aenor.es [ISO]

Norma ISO 9004-1 UNE (41 páginas) International Standardization Organization. Una Norma Española. http://www.aenor.es [ISO]

Norma ISO 8402 UNE (30 páginas) International Standars Organization. Una Norma Española http://www.aenor.es [ISO]

[NOVATICA 137] *NOVATICA.Número 137, Enero-Febrero 1999. Monográfico Calidad del Software / Software de calidad. Publicada por la Asociación de Técnicos en Informática (ATI). www.ati.es*

[Oskarsson 1996] Oskarsson Ö, Glass R.L. *An ISO 9000 approach to building Quality Software*. Prentice-Hall (1996)

[Piattini 1996] M.G. Piattini, J.A. Calvo-Manzano, J. Cerveza, y L. Fernández. *Análisis y diseño detallado de aplicaciones informáticas de gestión. RA-MA (1996).*

[Piattini 2001] M.G. Piattini, E. Del Peso (Editores). *Auditoría Informática. Un enfoque práctico.* 2ª Revisión ampliada y revisada. Ed. RA-MA (2001)

[Pressman 1993] R. S. Pressman. *Ingeniería del software. Un enfoque práctico*. 3ª Edición. McGrawHill (1993)

[Pressman 1998] R. S. Pressman. *Ingeniería del software. Un enfoque práctico*. 4ª Edición. McGrawHill (1998)

[Sanders 1994] J. Sanders, E. Curran. *Software Quality.* Addison-Wesley (1994)

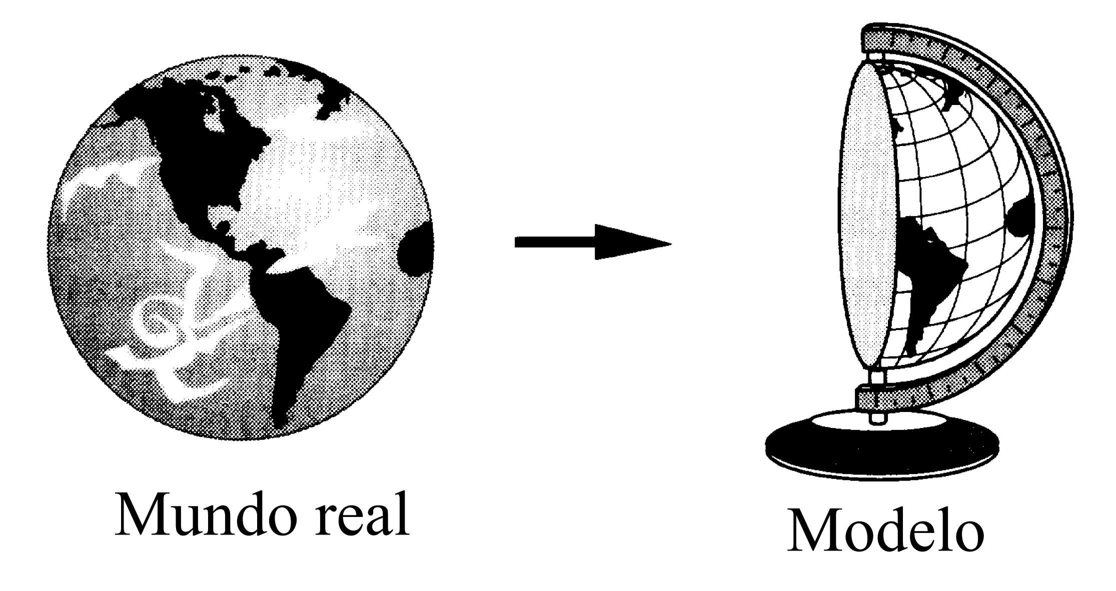
[SEI] Software Engineering Institute http://www.sei.cmu.edu

[Tingey 1997] M. O. Tingey. *Comparing ISO 9000, Malcom Baldrige and the SEI CMM for software.* Prentice-Hall (1997).

Tema 3º

Introducción a la Tecnología de Objetos

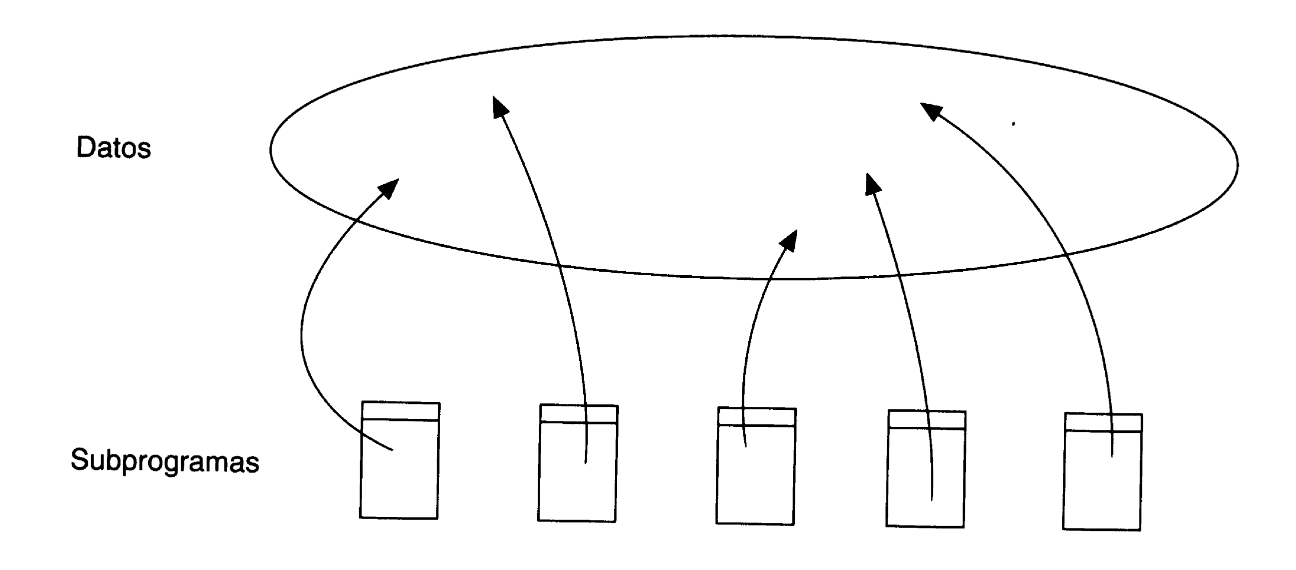
- La evolución del modelo de objetos
- Fundamentos del modelo de objetos
- Elementos del modelo de objetos
- Conceptos clave
- Resumen



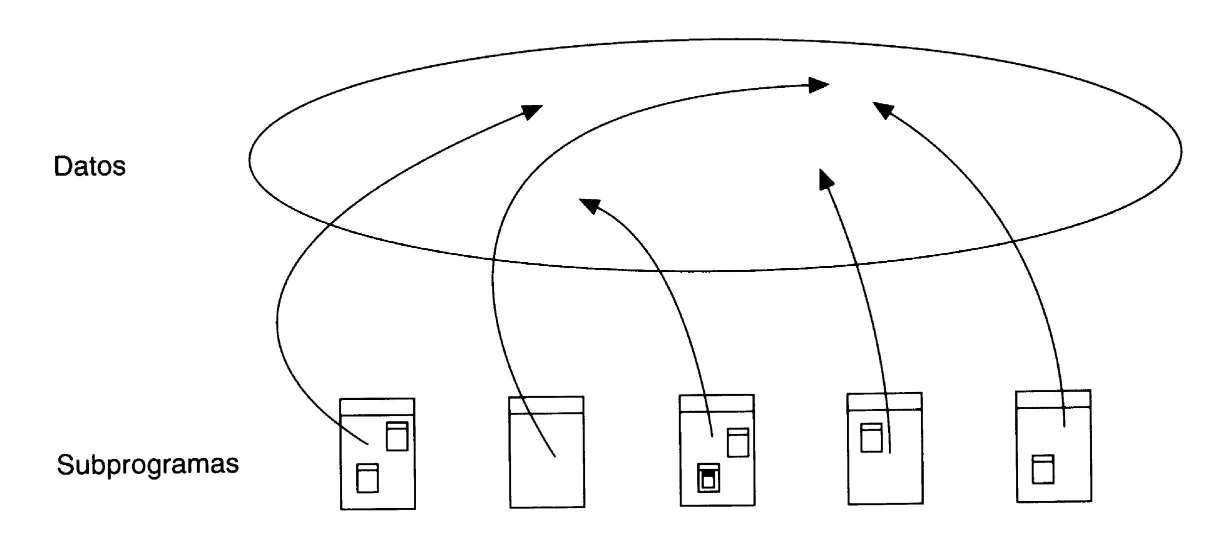
CONTENIDOS

3.1 La evolución del modelo de objetos

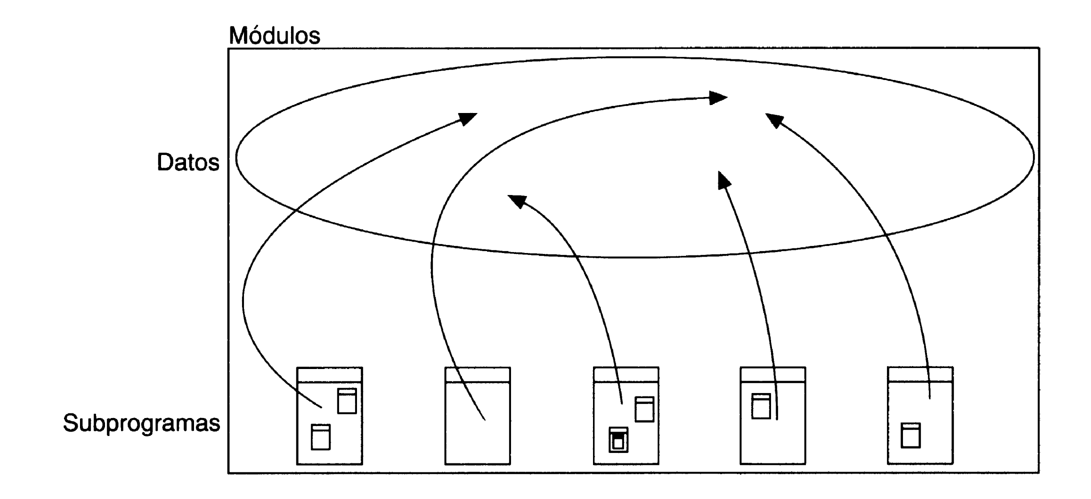
- **Tendencias en ingeniería del software** *[Booch 94]*
  - Hay una correspondencia entre la evolución de la ingeniería del software y la evolución de los lenguajes de alto nivel.
- **Topología de los lenguajes de la primera generación y principios de la segunda (FORTRAN y COBOL)**
  - **Datos globales y subprogramas**
    - Solo soportaban paso de parámetros por dirección
    - Un error en una parte de un programa puede tener un devastador efecto de propagación a través del resto del sistema, porque las estructuras de datos globales están expuestas al acceso de todos los subprogramas
    - Cuando se realizan modificaciones en un sistema grande, es difícil mantener la integridad del diseño original



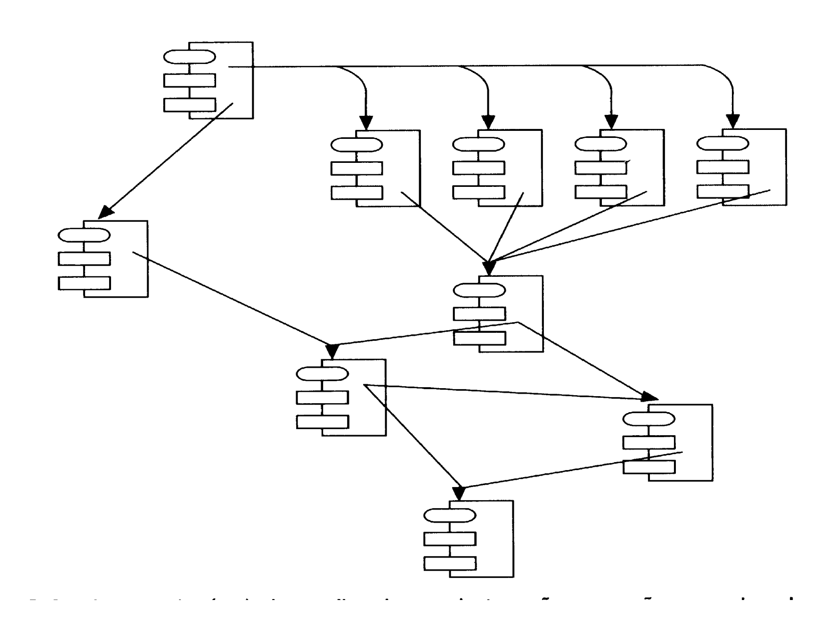
- **Topología de los lenguajes de finales de la segunda generación y principios de la tercera (PASCAL y C)**
  - **Nuevos mecanismos de paso de parámetros (paso por valor)**
  - **Se asentaron los fundamentos de la programación estructurada**
    - Los lenguajes permiten anidar subprogramas
    - Se permiten variables locales introduciendo los conceptos de ámbito y visibilidad de las declaraciones
    - Se avanza en las estructuras de control de flujo
  - **Surgen los métodos de diseño estructurado descendente**
    - Se construyen grandes programas utilizando los subprogramas como bloques físicos de construcción
  - Esta topología es una variante más refinada de la anterior mejorando el control sobre las abstracciones algorítmicas, pero sigue fallando a la hora de superar los problemas de la producción de software a gran escala.



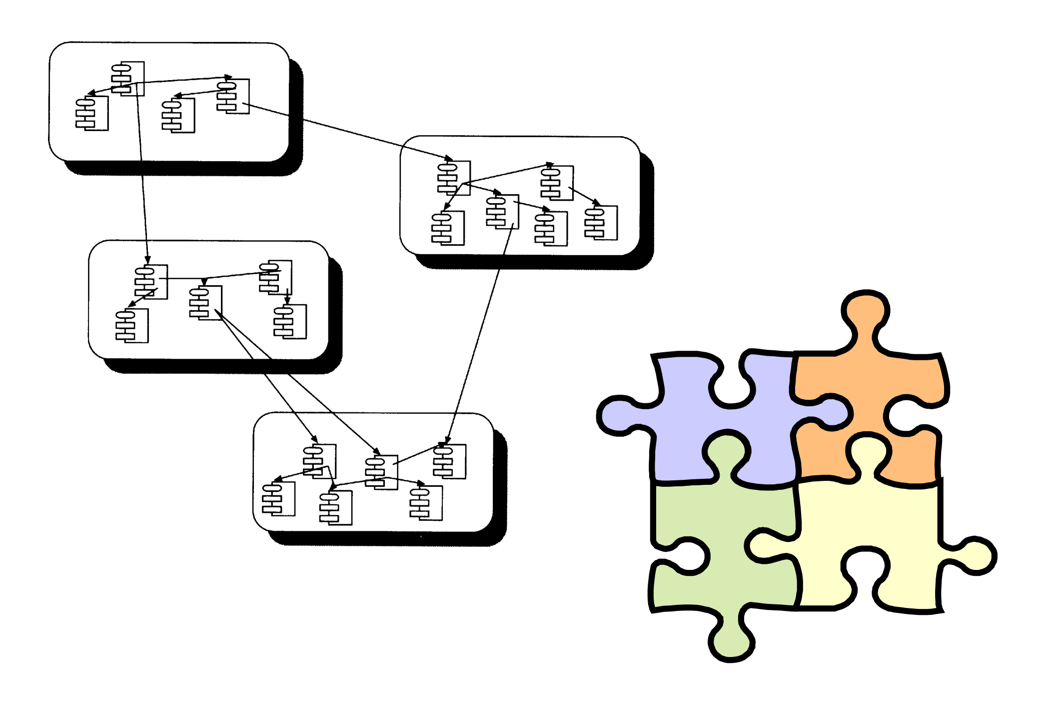
- **Topología de los lenguajes de finales de la tercera generación (Modula-2)**
  - **Introducen los módulos compilados separadamente**
    - Inicialmente son contenedores arbitrarios de datos y subprogramas
    - Aparecen extensiones para otros lenguajes (por ejemplo en Pascal se incorporan las units)
    - Tenían pocas reglas que exigiesen una consistencia semántica entre los interfaces de los módulos
  - **Desembocan en la definición de Tipos Abstractos de Datos (TAD)**
  - **Aparecen los métodos de diseño dirigido por los datos (Jackson, Warnier)**
    - Proporcionaron una aproximación disciplinada a los problemas de realizar abstracciones de datos en lenguajes orientados algoritmicamente



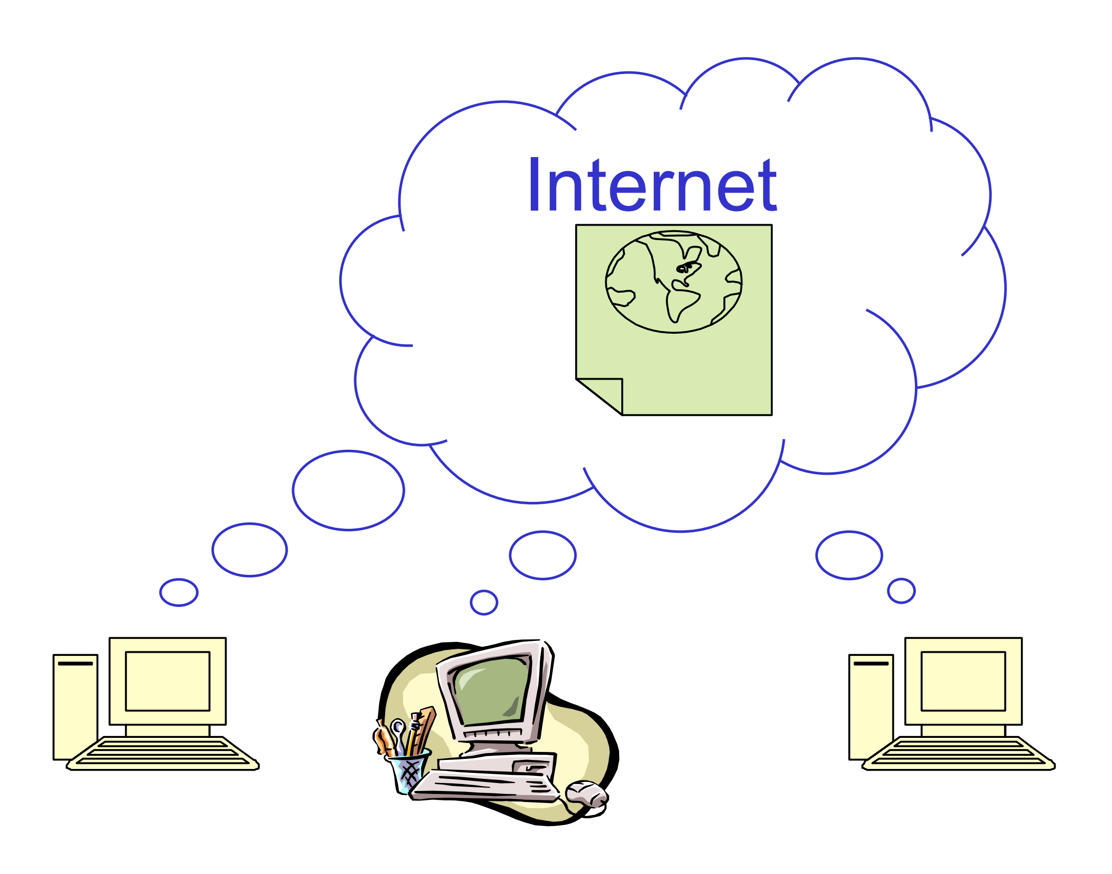
- **Topología de los lenguajes basados en objetos y orientados a objetos (ADA, Object Pascal, Smalltalk, Eiffel, C++, Java)**
  - **Introducen los conceptos de clase y objetos**
  - **Los lenguajes basados en objetos no tienen herencia ni polimorfismo**
  - **Los lenguajes orientados a objetos cumplen las características básicas del modelo de objetos**
  - **El bloque físico de construcción de estos lenguajes es el módulo, que contiene una agrupación de clases y objetos**
  - **En sistemas muy complejos las clases, objetos y módulos proporcionan un medio esencial, pero, aun así, insuficiente de abstracción**



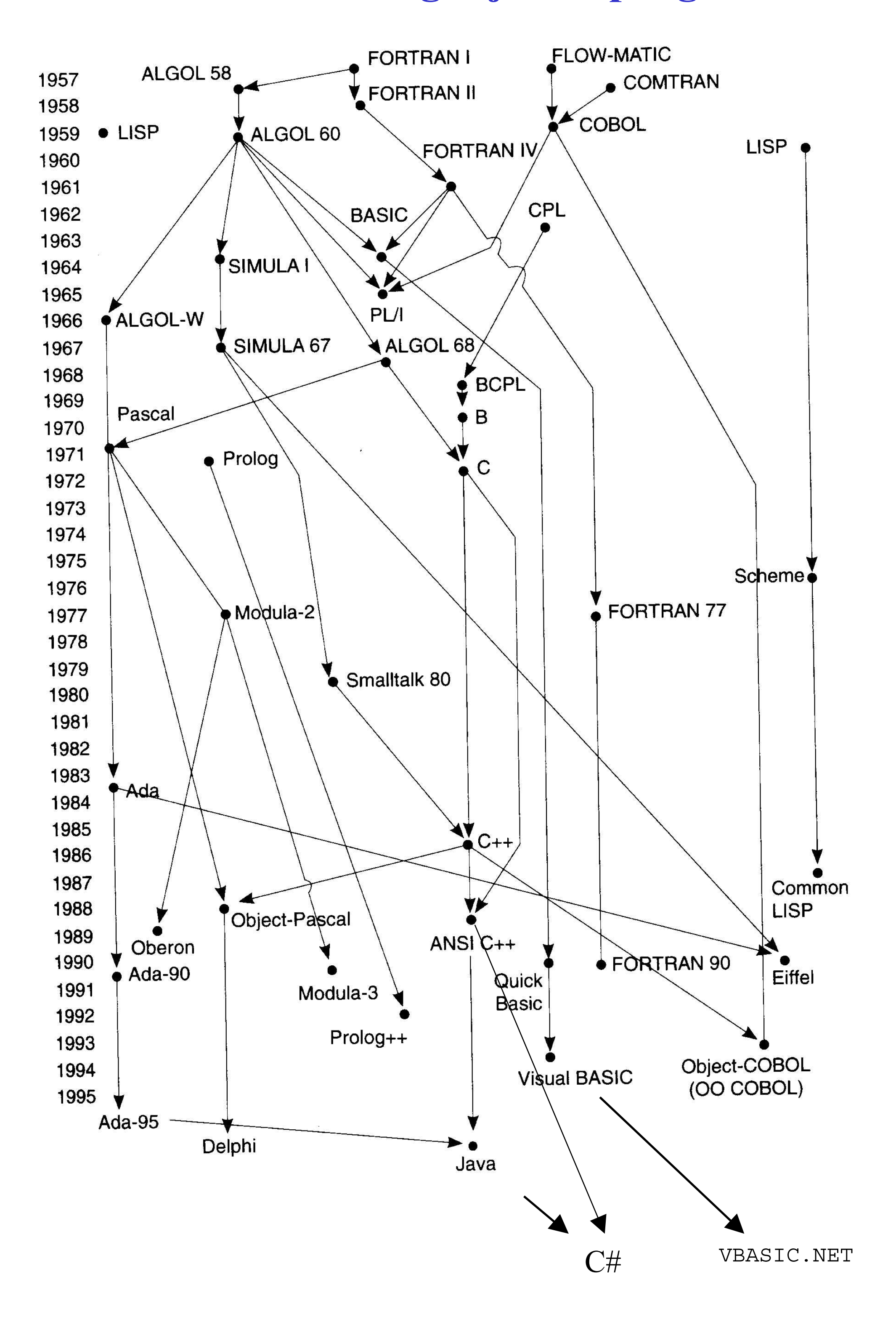
- **Topología de aplicaciones utilizando componentes (ActiveX, JavaBeans)**
  - **Sistemas operativos y lenguajes de programación soportan directamente el concepto de componente (Visual Basic, Delphi, C++Builder, Enterprise JavaBeans).**
  - **Un componente es una parte física y reemplazable de un sistema con un conjunto de interfaces y proporciona la implementación de dicho conjunto**
    - Un componente representa típicamente el empaquetamiento físico de diferentes elementos lógicos, como clases, interfaces y colaboraciones
  - **El desarrollo orientado a objetos proporciona la base fundamental para ensamblar sistemas a partir de componentes**



- **Ingeniería web**
  - **Lenguajes y especificaciones para aplicaciones en la web definidos por el *World Wide Web Consortium* (www.w3c.org)**
  - **Lenguajes de marcas: HTML, XML y SGML**
  - **Lenguajes de programación (Java y C#)**
  - **Tecnologías Java y .NET**
  - **Lenguajes de script (ASP, JavaScript, VisualBasicScript)**
  - **Applets y Servlets**
  - **Computación ubicua**
  - **Usabilidad**
  - **Servicios web**
  - **Gestión del Conocimiento**



Evolución de los lenguajes de programación



El lenguaje de programación C++

[Stroustrup 2000] [Stroustrup 1994]

- Lenguaje Orientado a Objetos híbrido
    - Mantiene compatibilidad en parte con el lenguaje C permitiendo el uso de funciones libres y de clases
    - Es un C mejorado
    - Soporta programación genérica
- Lenguaje compilado muy eficiente y rápido
- El código generado debe compilarse específicamente para cada sistema operativo y plataforma
- Es muy extenso y con bastantes peculiaridades
- Primer ejemplo en C++

```
// Hola.cpp
// Primer programa en C++
//  Emite el saludo "Hola a todos"

// incluye la biblioteca de Entrada/Salida
#include <iostream>
using namespace std;  //Para usar la biblioteca estándar
int main()
 {
  cout << "Hola a todos\n";
  return 0;
 }
```

- Para compilar con el compilador de GNU (www.gnu.org) en *linux*

`$ g++ -o Hola.out Hola.cpp`

- Para ejecutar el programa

`$./Hola.out`

http://gcc.gnu.org

El lenguaje de programación C++ ”Hola a todos” en C++ Builder Modo consola (I)

• C++ Builder es un compilador y un entorno de desarrollo de C++ con componentes. Pero también puede trabajar en modo consola (http://www.borland.com)

• Para realizar una aplicación en modo consola se deben seguir los siguientes pasos

  • Por defecto arranca en modo gráfico. Debe cerrarse todo de la siguiente forma: ir al menú **File** , al desplegarse se elige **Close All**

  • Queda todo vacío y comenzamos eligiendo en el menú **File** la opción **New...**

  • Se elige el modo **Console Wizard**

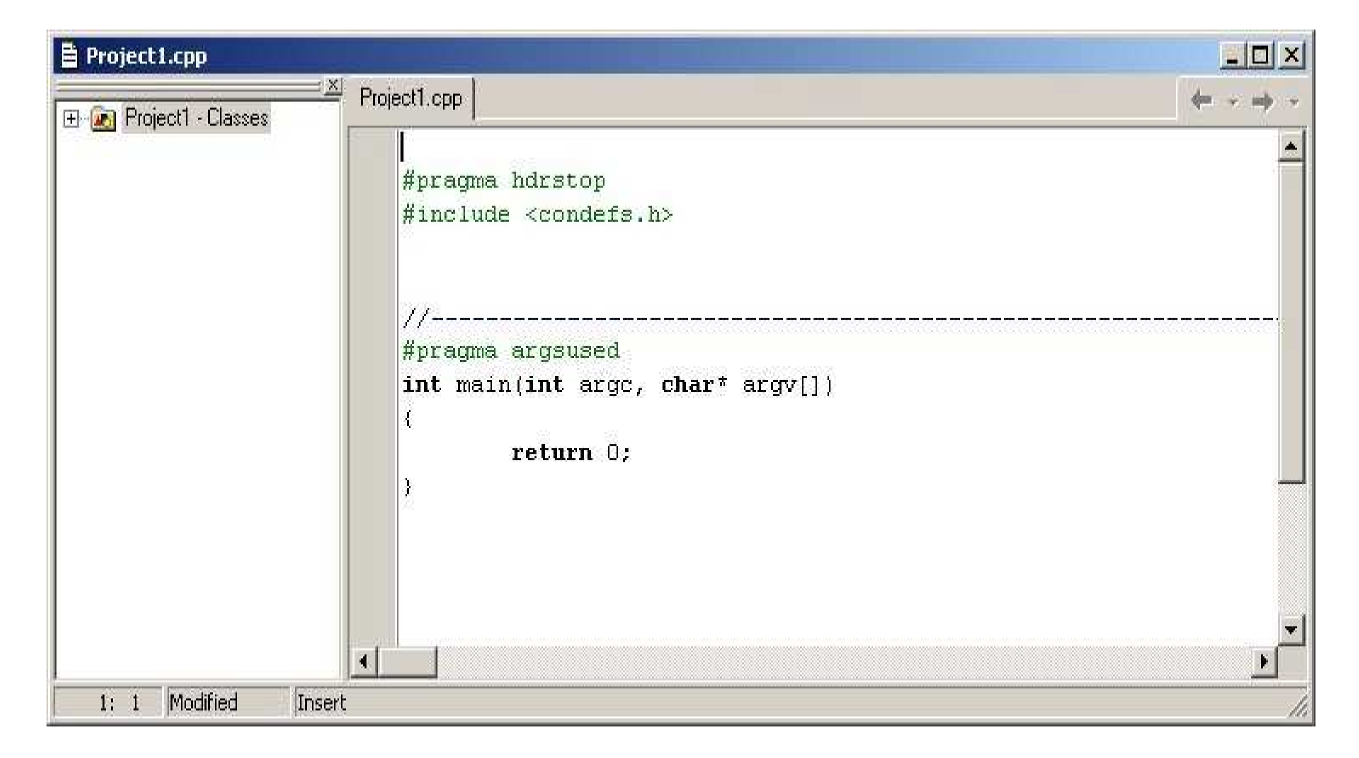
  • Sale un cuadro de diálogo con unos botones de radio. Se debe elegir:

    • Window Type: **Console**

    • Execution Type: **EXE**

    • No marcar el resto de opciones

  • Al aceptar aparece la ventana siguiente:



El lenguaje de programación C++ ”Hola a todos” en C++ Builder Modo consola (II)

- Ahí podemos escribir el siguiente programa, teniendo en cuenta que el propio entorno ya ha puesto unas líneas de código que respetamos.

```
#pragma hdrstop
#include <condefs.h>
//---------------------------
#include <iostream>
using namespace std;
//----------------------------
// Para el uso de getch() se incluye conio
#include<conio>
//----------------------------
#pragma argsused
int main(int argc, char* argv[])
{
        cout<<"Hola a todos\n";

        cout<<"Pulse una tecla para finalizar programa" ;
        getch(); //Espera que se pulse una tecla
        return 0;
}
```

El lenguaje de programación Java (I)

[Arnold 1998]

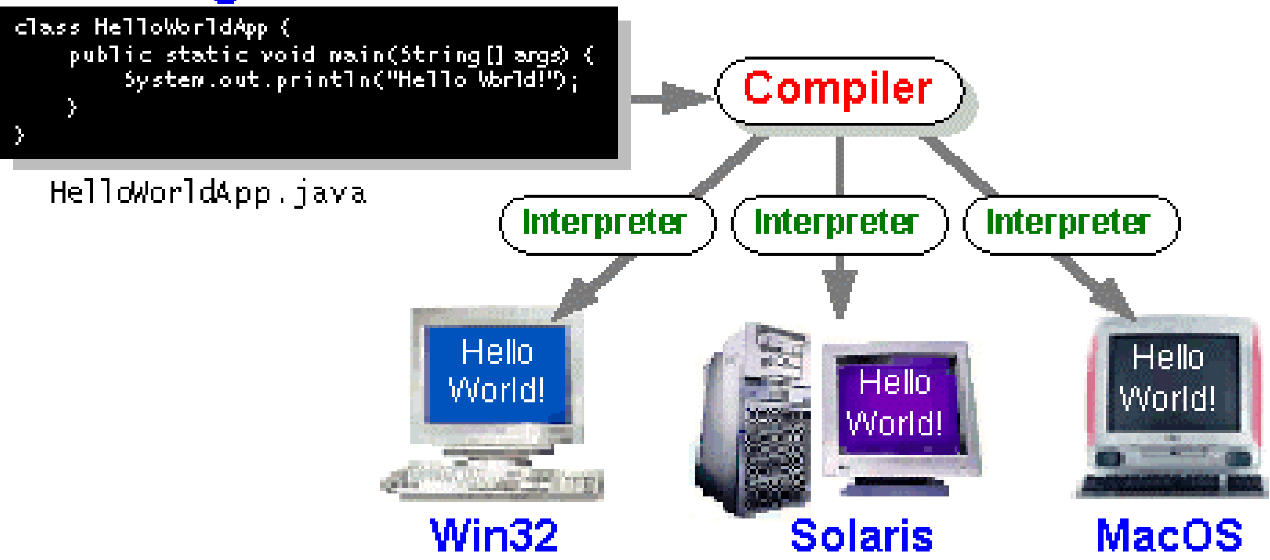
- Lenguaje Orientado a Objetos puro
- Todo el código se organiza en clases
- Se pueden ejecutar las clases independientemente, lo que facilita la prueba unitaria de cada clase
- Puede haber tantas funciones *main( )* como clases.
- Lenguaje interpretado independiente de plataforma y sistema operativo
- Relativamente pequeño y con una biblioteca de clases muy grande

```
/* HolaATodos.java
   Primer programa en Java
   Emite el saludo "Hola a todos"
*/
class HolaATodos {
   public static void main (String[] args){
        System.out.println( "Hola a todos" );
   }
}
```



DILBERT reprinted by permission of United Feature Syndicate Inc.

El lenguaje de programación Java (II) Compilando e interpretando



• Para compilar con el compilador de SDK que se puede obtener en **www.javasoft.com**

```
$ javac HolaATodos.java
```

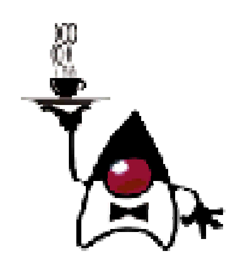
• Genera un fichero denominado `HolaATodos.class`

• Este fichero está en un formato binario denominado bytecode

• Los Bycodes están en un lenguaje intermedio que ejecuta la máquina virtual de java (JVM), Java Virtual Machine)

• Para ejecutar el programa se interpreta el fichero `HolaATodos.class`

```
$ java HolaATodos
```



El lenguaje de programación Java (III)
Ejemplos sencillos

• **Ejemplo 1**

```
class PruebaString{
          public static void main (String[] arguments){
                    String saludo="Hola";
                    System.out.println(saludo + " a todos");
          }
}
```

• **Ejemplo 2**

```
class PruebaString2{
          public static void main (String[] arguments){
                    String saludo="Hola";
                    System.out.println(saludo + " a todos");
                    int  valor =5;
                    System.out.println("valor="+ valor+'.');
          }
}
```

• **Ejemplo 3**

```
/**
Tabla de multiplicar del 8
Ilustra el bucle while (mientras que)
*/

class BucleWhile {
   public static void main (String[] args){
          int i=1;
          int tabla=8;
          while (i<11)
                    System.out.println(" "+i+'x'+tabla+'='+i++*tabla);
   }
}
```

El lenguaje de programación Java (IV)
Ejemplos sencillos

- **Ejemplo 4**

```
// Entrada y salida
import java.io.*;

class PruebaString3{
          public static void main (String[] arguments)
                                    throws IOException{
                    BufferedReader entrada= new BufferedReader
                                (new InputStreamReader(System.in));

                    System.out.println("¿Cómo te llamas?");
                    String nombre= entrada.readLine();
                    System.out.println("Hola "+ nombre);
          }
}
```

- **Ejemplo 5**

```
/**
Tabla de multiplicar
Ilustra el bucle for (para) y conversión de String a entero
*/
import java.io.*;

class TablaMultiplicar{
  public static void main (String[] arguments) throws IOException{
  BufferedReader entrada= new BufferedReader
                              (new InputStreamReader(System.in));

System.out.println
("Hola, dime de que numero hacemos la tabla de multiplicar");

String cadena= entrada.readLine();

int tabla=(new Integer (cadena)).intValue();

for (int i=1;i<11;i++)
   System.out.println(" "+i+'x'+tabla+'='+i*tabla);
          }
}
```

El lenguaje de programación Java (V)
Ejemplos sencillos

- **Ejemplo 6**

```
/**
Tabla de multiplicar números reales
Ilustra el bucle for (para) y conversion de string a real

@author Juan Manuel Cueva Lovelle
*/



import java.io.*;

class TablaMultiplicarReales{
 public static void main (String[] arguments) throws IOException{

  BufferedReader entrada= new BufferedReader
                                (new InputStreamReader(System.in));

 System.out.println
   ("Hola, dime de que numero REAL hacemos la tabla de multiplicar");


String cadena= entrada.readLine();

double tabla=(new Double (cadena)).doubleValue();

for (int i=1;i<11;i++)

          System.out.println(" "+i+'x'+tabla+'='+i*tabla);
          }
}
```

Interacción entre Java y HTML (I)

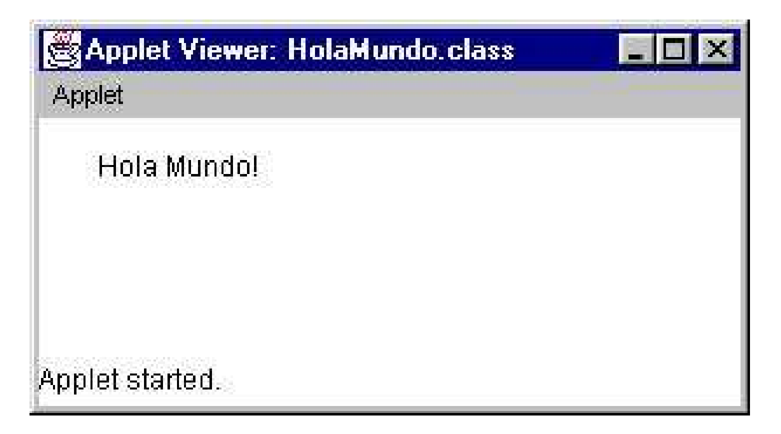
- Un applet es una pequeña aplicación accesible en un servidor Internet, que se transporta por la red, se instala automáticamente y se ejecuta in situ como parte de un documento web
- Un applet es una mínima aplicación Java diseñada para ejecutarse en un navegador Web
- El applet asume que el código se está ejecutando desde dentro de un navegador
- Los applets no tienen el método *main()* se cargan por el navegador desde un fichero en lenguaje HTML
- HTML es la abreviatura *HyperText Markup Language* que significa Lenguaje de marcas para hipertexto, y es el lenguaje utilizado para crear páginas web.
- Es decir para ejecutar un applet es necesario crea un fichero HTML y cargarlo en el navegador.
- El fichero mínimo HTML para cargar un applet es el siguiente:

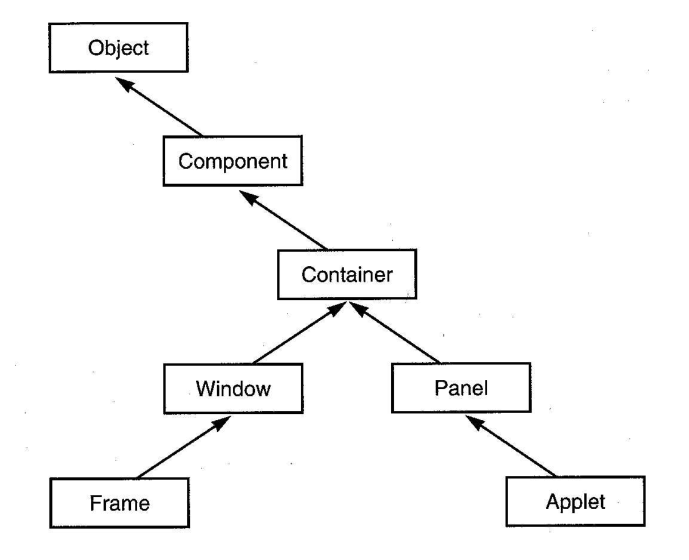
```
<HTML>
<APPLET CODE=HolaMundo.class WIDTH=300 HEIGHT=100>
</APPLET>
</HTML>
```

- El JDK proporciona un visualizador de applets denominado *appletviewer*

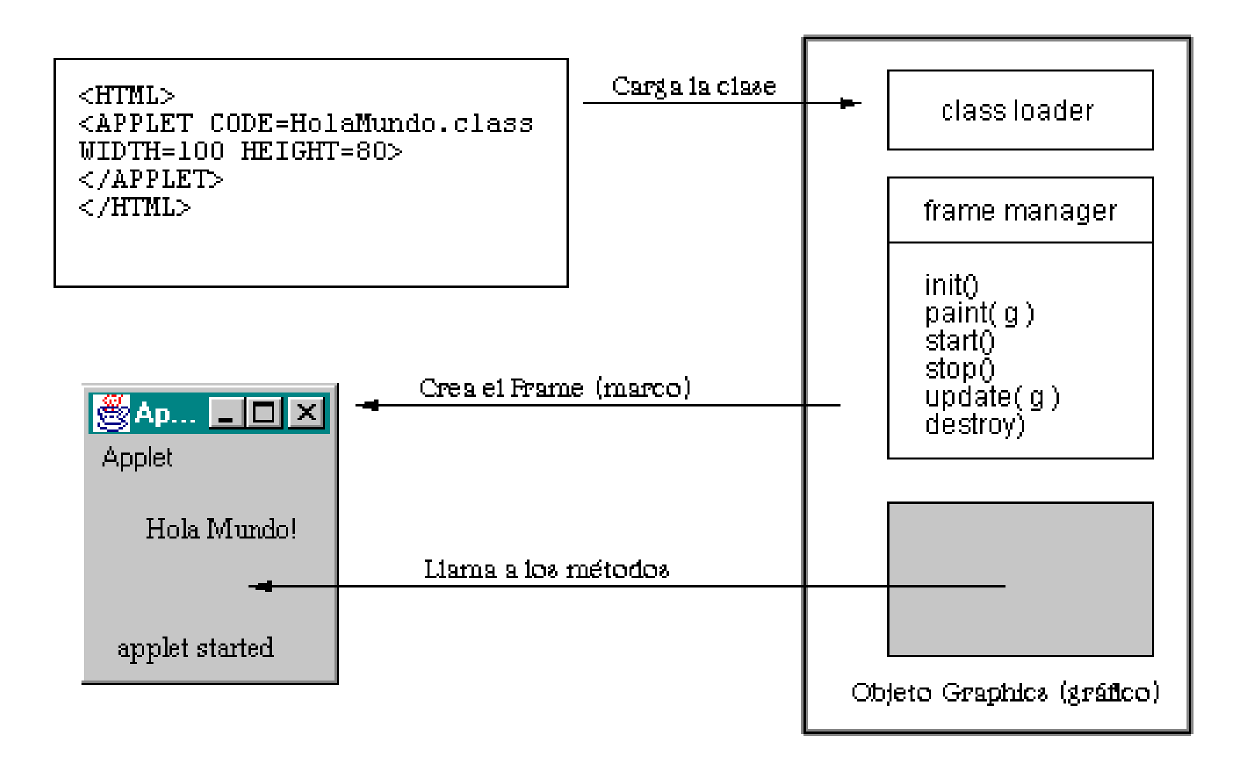
Interacción entre Java y HTML (II)

```
// Applet mínimo HolaMundo
import java.awt.Graphics;
import java.applet.Applet;
public class HolaMundo extends Applet {
  public void paint( Graphics g ) {
        g.drawString( "Hola Mundo!” ,25,25 ) ;
        }
}
```





Interacción entre Java y HTML (III)



Interacción entre Java y HTML (IV) Ejemplo 2

```
<HTML>
  <HEAD>
  <TITLE> Una prueba de un Applet </TITLE>
  </HEAD>
  <BODY>

      Aquí se ejecuta el applet

  <APPLET CODE="AppletString3.class" WIDTH=300 HEIGHT=100>
  </APPLET>
  </BODY>
  </HTML>
```



Interacción entre Java y HTML (V) Ejemplo 2

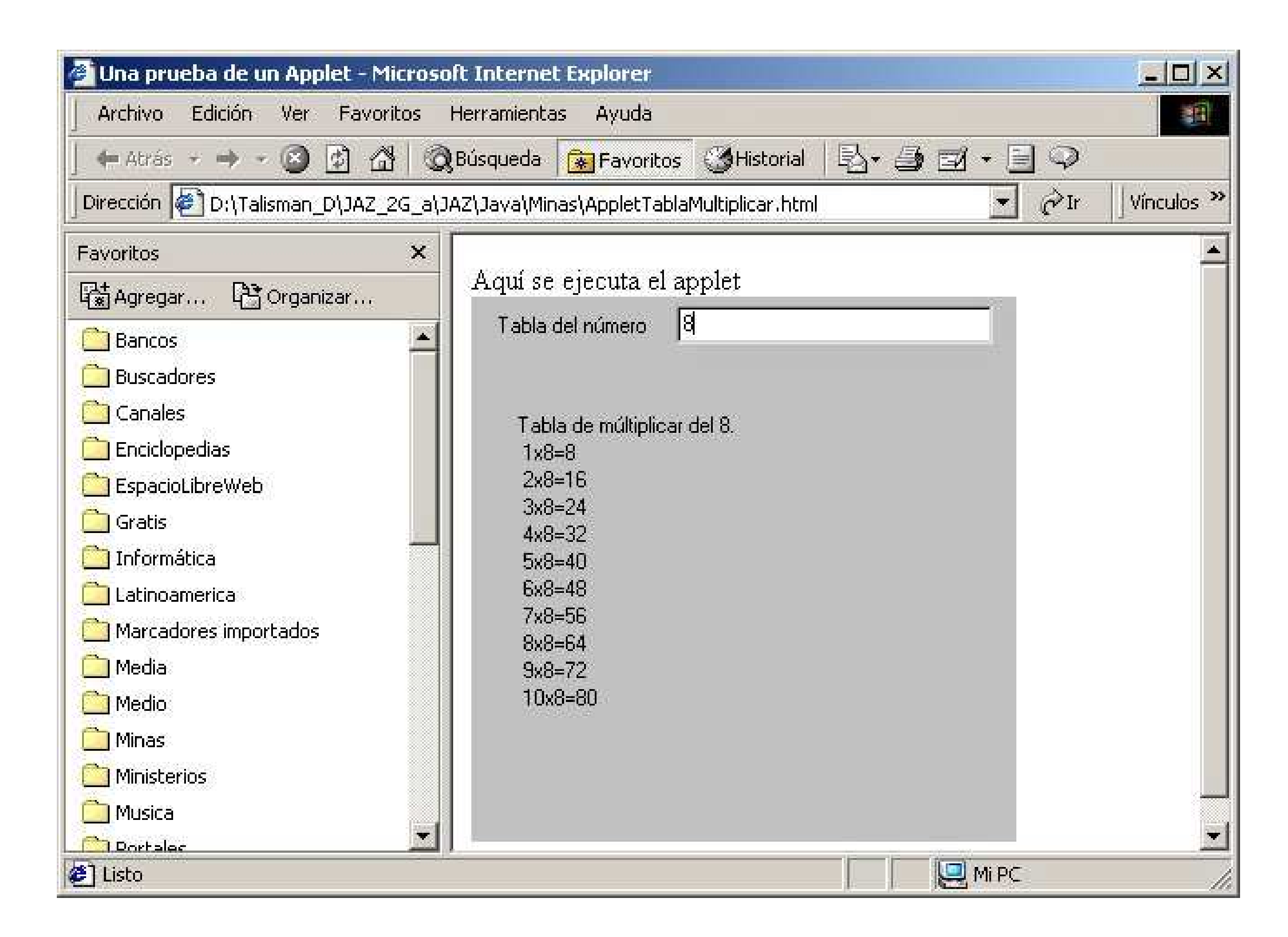
```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class AppletString3 extends Applet
          implements ActionListener {
          Label etiNombre =new Label ("Nombre");
          TextField campoNombre =
                    new TextField("Valor por defecto",25);
          String nombre;
public void init(){
                    add (etiNombre);
                    add (campoNombre);
                    campoNombre.addActionListener(this);
            }
public void actionPerformed (ActionEvent e) {
           nombre=campoNombre.getText();
           repaint();
          }

public void paint (Graphics g) {
    g.drawString("Hola "+ nombre + '.', 25, 75);
    }
}
```

Interacción entre Java y HTML (VI) Ejemplo 3

```
<HTML>
  <HEAD>
  <TITLE> Una prueba de un Applet </TITLE>
  </HEAD>
  <BODY>
      Aquí se ejecuta el applet
  <APPLET CODE="AppletTablaMultiplicar.class" WIDTH=300 HEIGHT=300>
  </APPLET>
  </BODY>
  </HTML>
```



Interacción entre Java y HTML (VII)
Ejemplo 3

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class AppletTablaMultiplicar extends Applet
                    implements ActionListener {
          Label etiNumero = new Label ("Tabla del número");
          TextField campoNumero = new TextField(25);
          String numero;


          public void init(){
                    add (etiNumero);
                    add (campoNumero);
                    campoNumero.addActionListener(this);
            }

          public void actionPerformed (ActionEvent e) {
           numero=campoNumero.getText();
           repaint();
          }

  public void paint (Graphics g) {
    g.drawString("Tabla de múltiplicar del "+ numero + '.', 25, 75);
    int tabla=(new Integer (numero)).intValue();

    for (int i=1;i<11;i++)
          g.drawString(" "+i+'x'+tabla+'='+i*tabla, 25, 75+i*15);
    }
}
```

El lenguaje C#

El primer programa

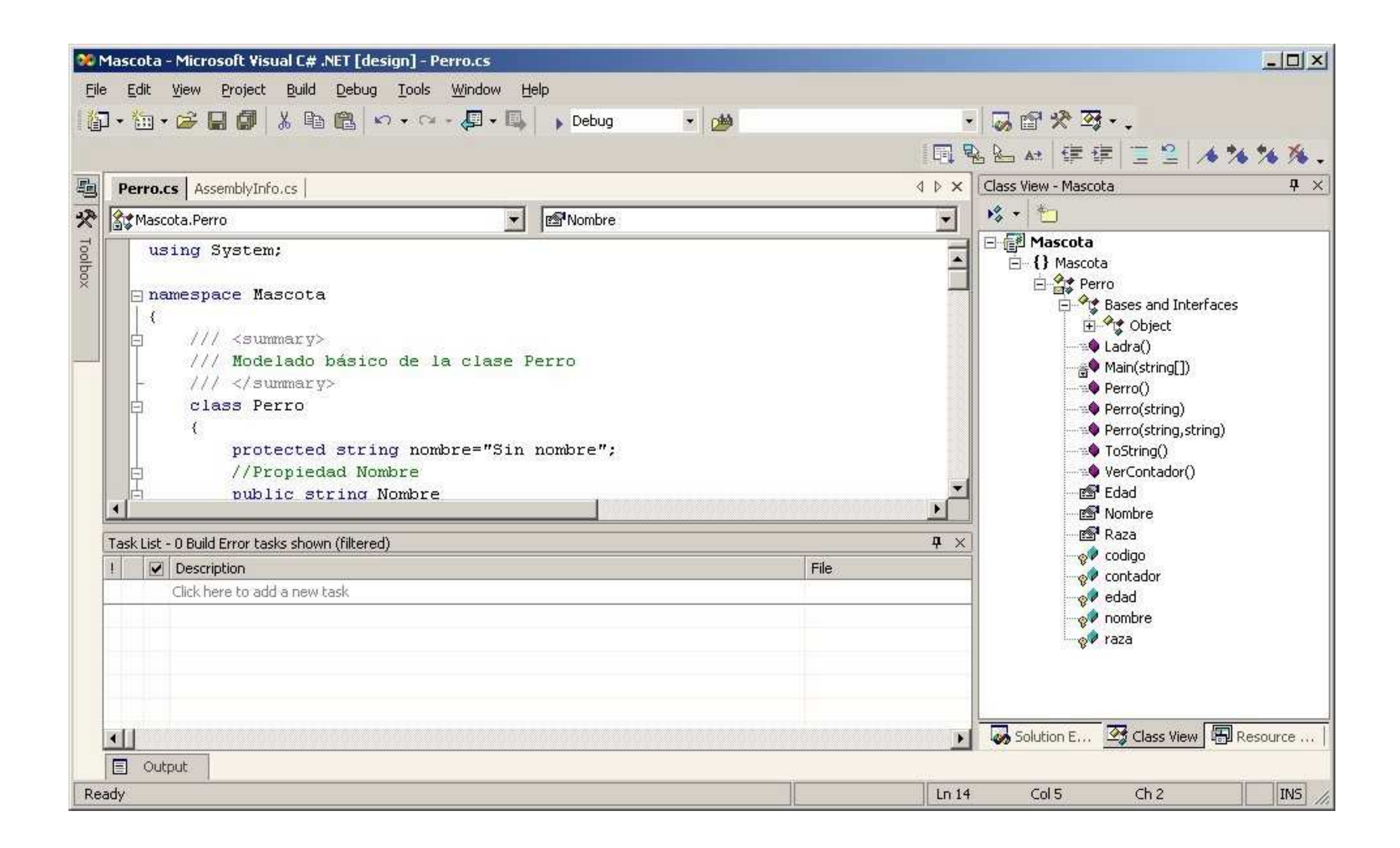
```
// Hola.cs
// Versión 1.0 6-Enero-2003
// Autor: Juan Manuel Cueva Lovelle
// Primer programa en C#

using System;

namespace Saludo
{
    /// <summary>
    /// La clase Hola saluda en la consola
    /// </summary>
    class Hola
    {
          /// <summary>
          /// El método Main de Hola
          /// </summary>
          [STAThread]
          static void Main(string[] args)
          {
           Console.WriteLine("Hola a todos");
           //Espera que se pulse una tecla
           Console.ReadLine();
           } //Fin del método Main
    } //Fin de la clase Hola
} //Fin del namespace
```

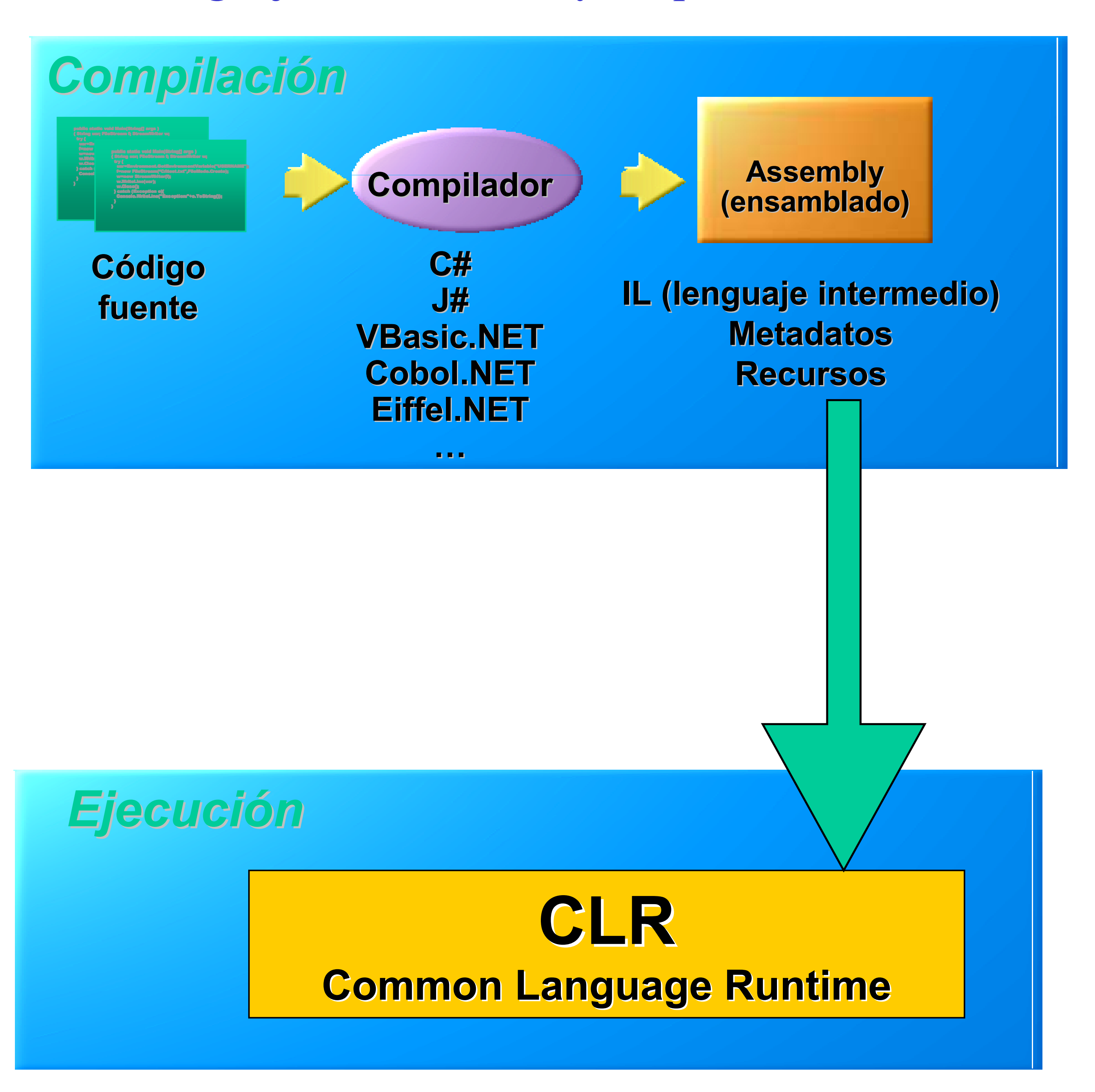
Crear , compilar y ejecutar C#
con Visual Studio.NET en modo consola

1. Iniciar Visual Studio.NET
2. Elegir **New Project**
3. Elegir **Visual C# Projects**
4. Seleccionar **Console Application**
5. Escribir el nombre del proyecto (**Name**)
6. Escribir el lugar donde se almacenará (**Location**)
7. Hacer click en **aceptar (OK)**
8. Aparece un esqueleto de código C#
9. Escribir el código del programa
10. Para compilar: en el menú **Build** elegir **Build Solution** (Ctrl+Shift+B)
11. Para ejecutar: en el menú **Debug** elegir **Start** (F5)



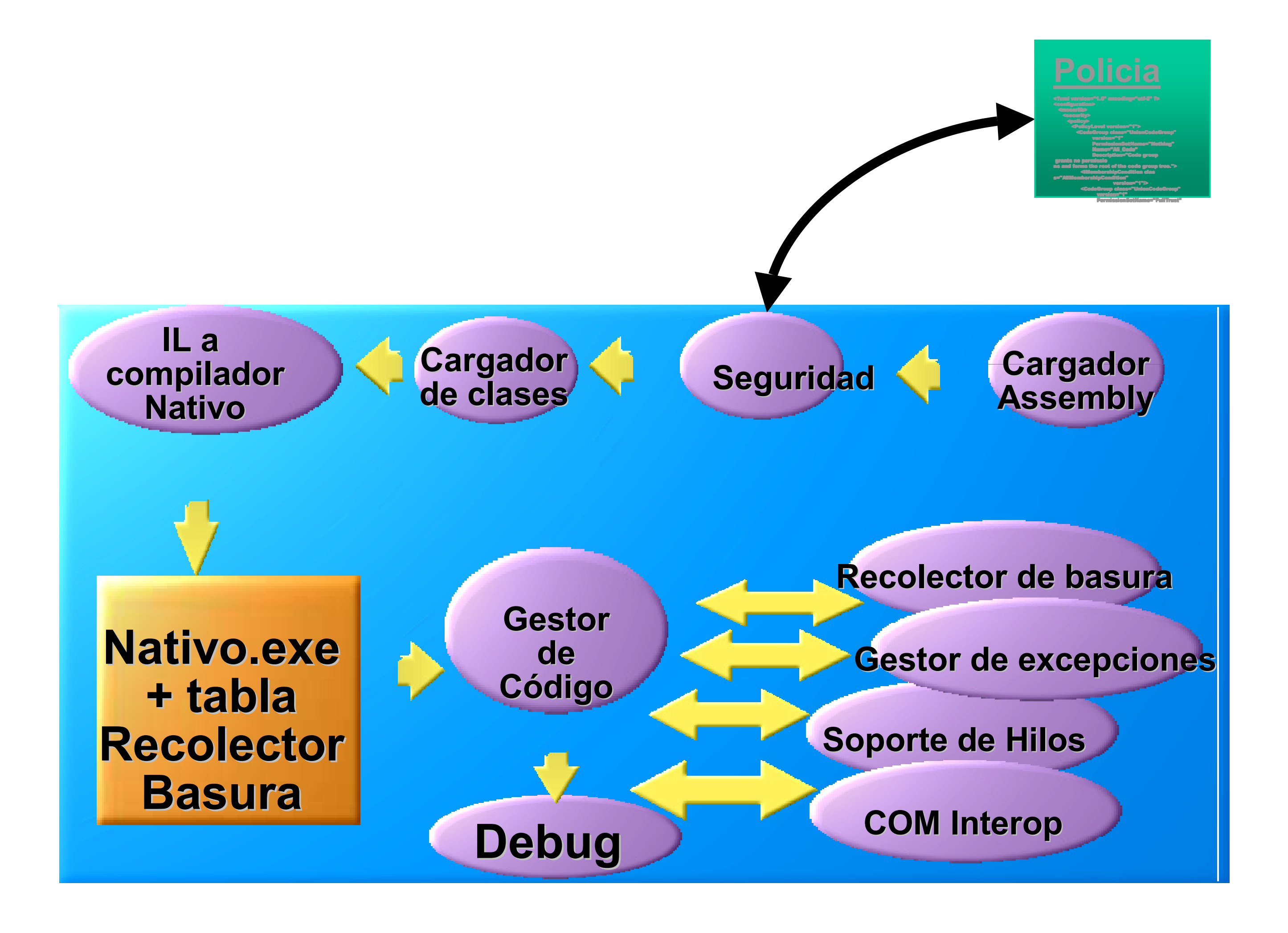
Entorno de desarrollo .NET

Lenguaje intermedio IL y máquina virtual CLR



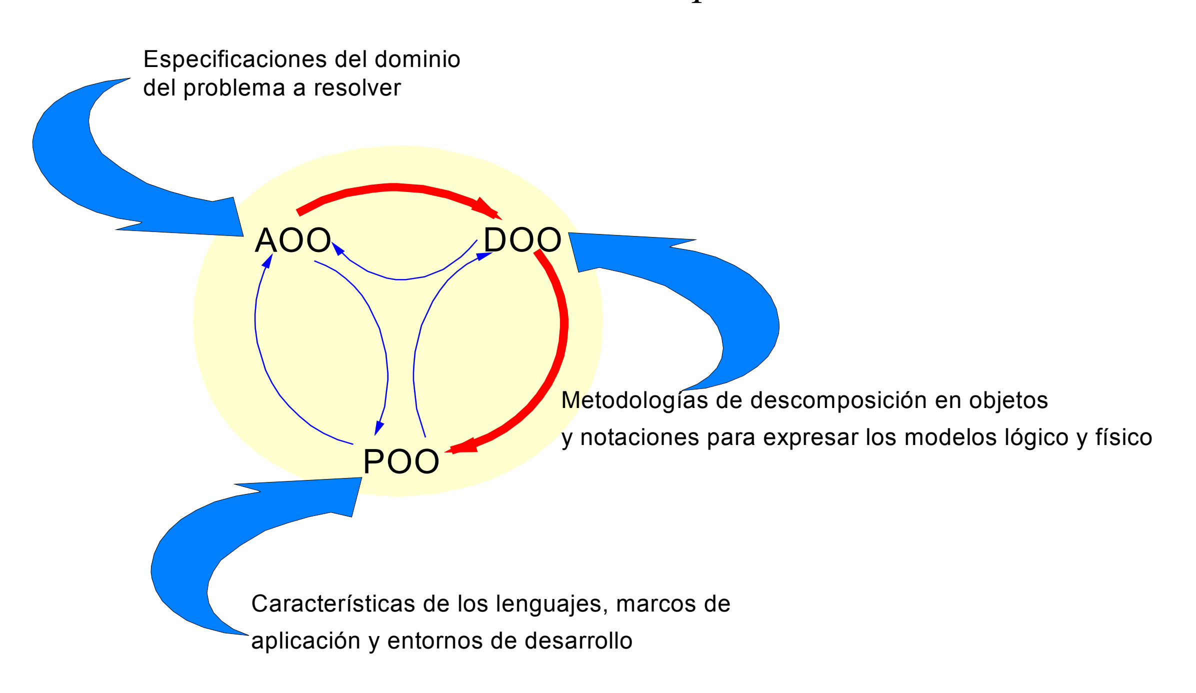
Máquina Virtual CLR

Common Language Runtime



3.2 Fundamentos del modelo de objetos

- **Programación orientada a objetos (POO)** *[Booch 94]*
  - *es un método de implementación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de alguna clase, y cuyas clases son, todas ellas, miembros de una jerarquía de clases unidas mediante relaciones de herencia*
- **Diseño orientado a objetos (DOO)** *[Booch 94]*
  - *es un método de diseño que abarca el proceso de descomposición orientada a objetos y una notación para describir los modelos lógico y físico, así como los modelos estático y dinámico del sistema que se diseña*
- **Análisis orientado a objetos (AOO)** *[Booch 94]*
  - *es un método de análisis que examina los requisitos desde la perspectiva de las clases y objetos que se encuentran en el vocabulario del dominio del problema*



3.2.1 Análisis y diseño orientado a objetos

Análisis Orientado a Objetos

Esta orientado a definir el problema, intentando el mejor entendimiento con el cliente

Preguntas que se deben plantear

- ¿Cuál es el comportamiento que se desea en el sistema?
- ¿Qué objetos existen en el sistema?
- ¿Cuáles son las misiones de los objetos para llevar a cabo el comportamiento deseado del sistema?

Diseño Orientado a Objetos

Esta orientado a construir un modelo que permita razonar sobre el problema y guiar la implementación

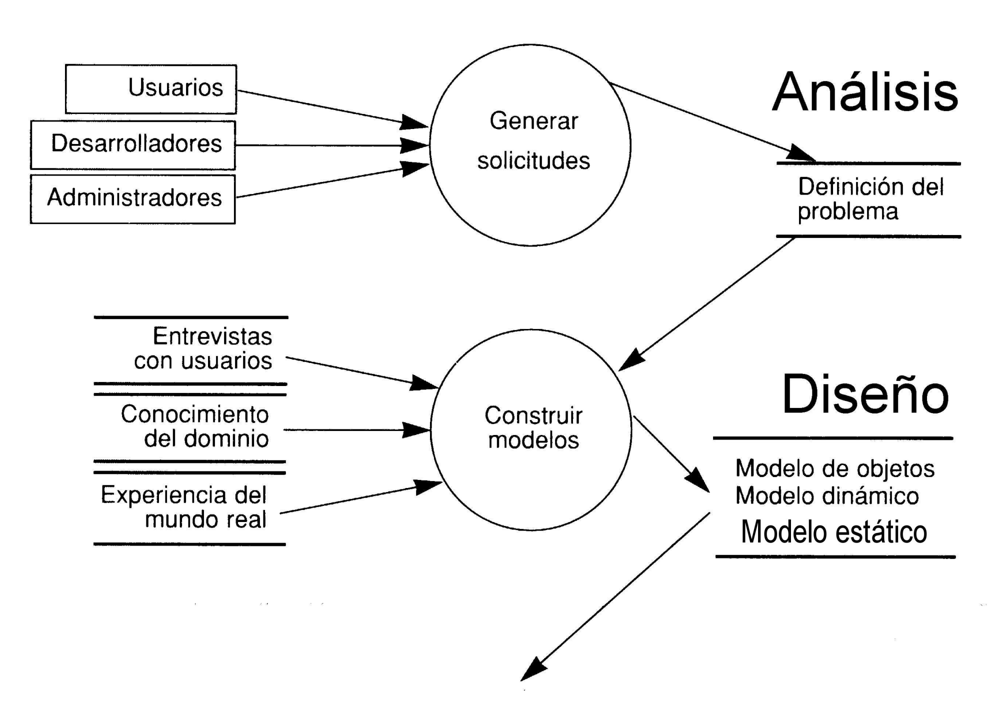
Preguntas que se deben plantear

Modelo lógico

- ¿Qué clases existen y como se relacionan estas clases?
- ¿Qué mecanismos se utilizan para regular la forma en que los objetos colaboran?

Modelo físico

- ¿Dónde debería declararse y construirse cada clase y objeto?
- ¿A qué procesador debería asignarse un proceso, y para un procesador dado, cómo deberían planificarse sus múltiples procesos?



3.2.2 Método y metodología

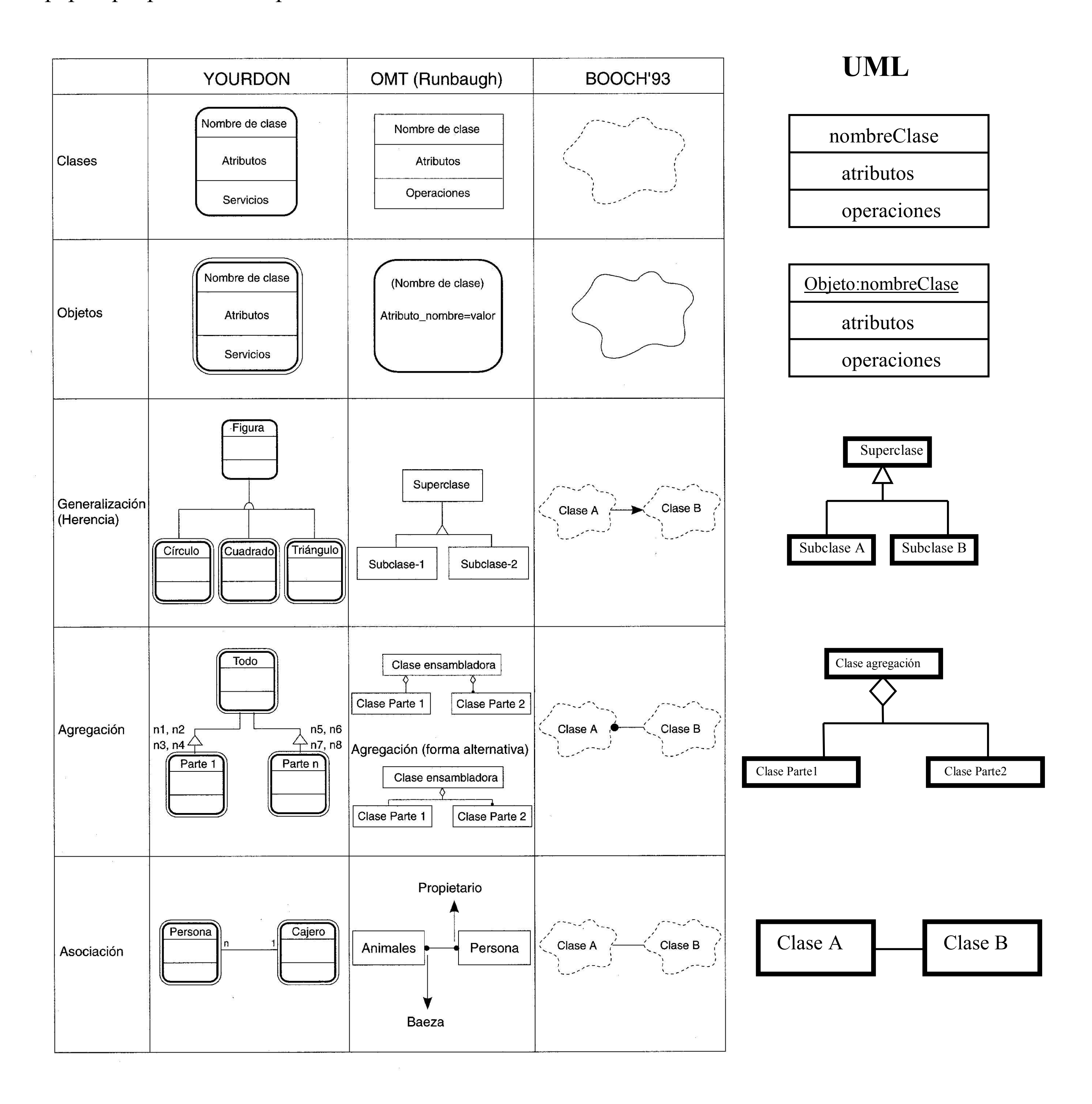
- *"Un **método** es un proceso disciplinado para generar un conjunto de modelos que describen varios aspectos de un sistema de software en desarrollo, utilizando alguna notación bien definida" [Booch 94]*
- *"Una **metodología** es una colección de métodos aplicados a lo largo del ciclo de vida del desarrollo de software y unificados por alguna aproximación general o filosófica" [Booch 94]*
  - Cada metodología puede catalogarse en uno de los grupos siguientes:
    - Diseño estructurado descendente
      - Yourdon y Constantine
      - Wirth
      - Dahl, Dijkstra y Hoare
    - Diseño dirigido por datos
      - Jackson
      - Warnier y Orr
    - Diseño orientado a objetos son las que siguen el modelo de objetos
      - Booch
      - OMT (Rumbaugh et al.)
      - Objectory (Jacobson et al.)
      - Schlaer-Mellor
      - Coad/Yourdon
      - Fusion (Coleman et al.)
      - Métrica - 3

3.2.3 Notaciones

- *"Una **notación** es un conjunto de diagramas normalizados que posibilita al analista o desarrollador el describir el comportamiento del sistema (análisis) y los detalles de una arquitectura (diseño) de forma no ambigua" [Booch 94]*
  - La acción de dibujar un diagrama no constituye ni análisis ni diseño
  - Una notación no es un fin por si misma
  - El hecho de que una notación sea detallada no significa que todos sus aspectos deban ser utilizados en todas las ocasiones
  - La notación son como los planos para un arquitecto o un ingeniero
  - Una notación no es más que un vehículo para capturar los razonamientos acerca del comportamiento y la arquitectura de un sistema
  - Las notaciones deben ser lo más independientes posibles de los lenguajes de programación, sin embargo facilita el proceso de desarrollo que exista una equivalencia entre la notación y los lenguajes de programación

3.2.4 Evolución de las notaciones

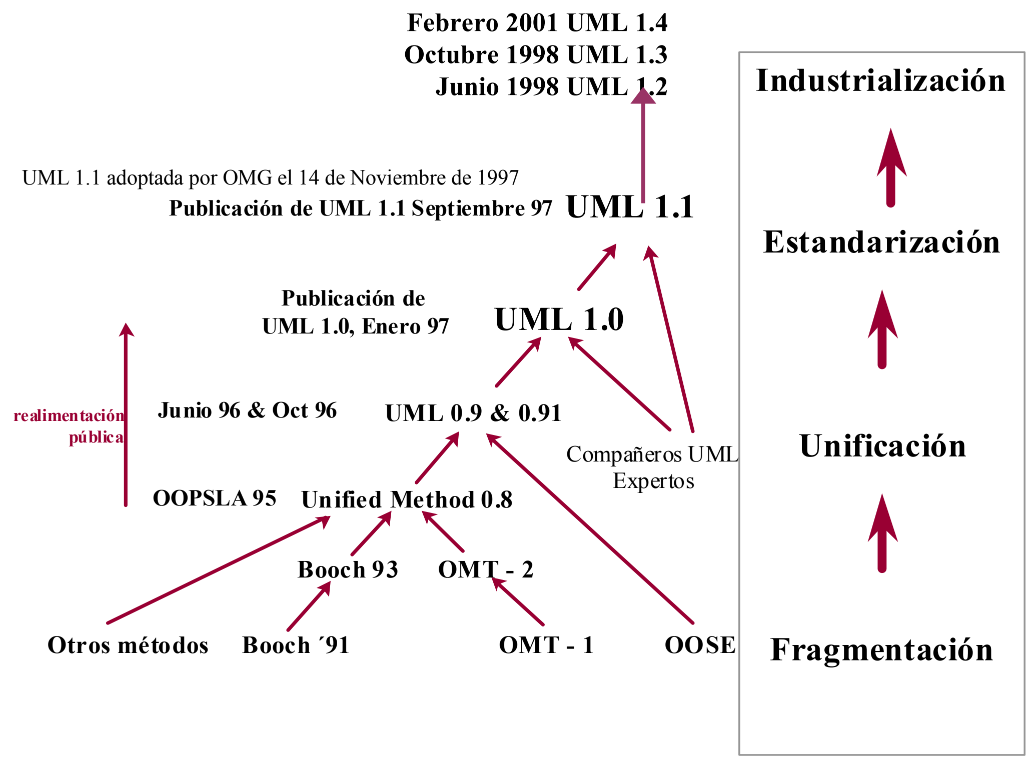
• La popularidad de las tecnologías de objetos ha generado docenas de metodologías de A y DOO
• Cada metodología tenia su propia notación
• La tendencia actual es separar la metodología de la notación y adoptar una notación estándar, papel que pretende adoptar UML



3.2.5 Unified Modeling Language (UML)

[Rational] [Booch 1999] [OMG]

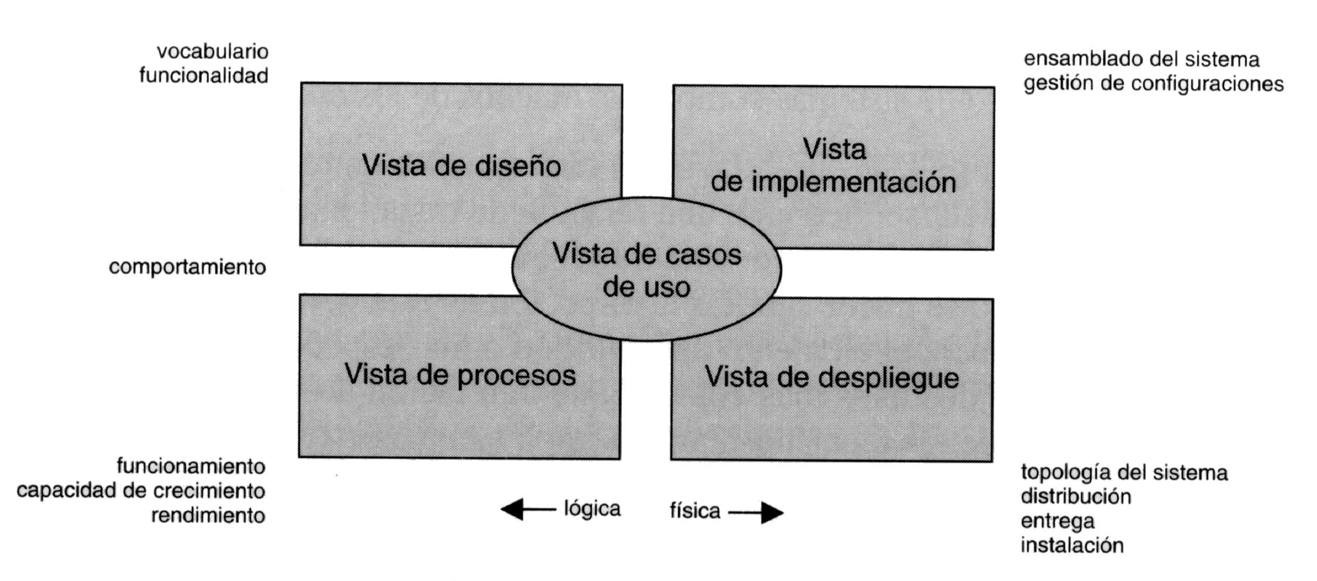
- UML es el sucesor de la ola de notaciones de A y DOO que aparecieron a finales de los 80 y principios de los 90
- UML unifica principalmente las notaciones de Booch, Rumbaught (OMT) y Jacobson. Pero pretende dar una visión más amplia de los mismos
- UML está definido como estándar por el OMG (Object Management Group) [OMG]
- UML es un **lenguaje de modelado**, no un método ni una metodología.
- UML es un lenguaje para
  - Visualizar
  - Especificar
  - Construir
  - Documentar
- Un método incluye
  - ***Lenguaje de modelado***: *Es la notación (en su mayoría gráfica) que utilizan los métodos para expresar los diseños.*
  - ***Proceso***: *Son los pasos que se aconsejan dar para realizar un diseño*



3.2.6 Modelado orientado a objetos

[Booch 99, capítulos 1y 2]

- Un modelo es una representación simplificada de la realidad
- Construimos modelos para comprender mejor el sistema que estamos desarrollando
- Es necesario observar el modelo desde distintas vistas, que tiene que ser compatibles entre sí
  - Al igual que en un plano se tienen varias perspectivas (planta, alzado, perfiles) compatibles entre sí.
- La arquitectura de un sistema orientado a objetos puede describirse con cinco vistas interrelacionadas
  - vista de casos de uso, vista de diseño, vista de procesos, vista de implementación, vista de despliegue
- Cada vista del modelo que resulta del diseño orientado a objetos es un conjunto de diagramas
- Estos diagramas permiten razonar sobre el comportamiento del sistema
- Cuando el modelo es estable cada uno de los diagramas permanece semánticamente consistente con el resto de los diagramas



Modelado de la arquitectura de un sistema.

3.2.7 Arquitectura de sistemas orientados a objetos

[Booch 99, capítulos 1,2, 31]

- La arquitectura de un sistema orientado a objetos puede describirse con cinco vistas interrelacionadas
  - Vista de casos de uso
    - Muestra los requisitos del sistema tal como es percibido por los usuarios finales, analistas, y encargados de pruebas.
  - Vista de diseño
    - Captura el vocabulario del espacio del problema y del espacio de la solución
  - Vista de procesos
    - Modela la distribución de los procesos e hilos (*threads*)
  - Vista de implementación
    - Modela los componentes y archivos que se utilizan para ensamblar y hacer disponible el sistema físico.
  - Vista de despliegue
    - Modela los nodos de la topología hardware sobre la que se ejecuta el sistema
- Cada una de estas vistas representan los planos del sistema
- Según la naturaleza del problema unas vistas tendrán más peso que otras. Así por ejemplo:
  - Los sistemas con grandes cantidades de datos dominaran las vistas de diseño
  - Los sistemas con uso intensivo de interfaces gráficas de usuario dominarán las vistas de casos de uso
  - Los sistemas de tiempo real las vistas de procesos tienden a ser las más importantes
  - En los sistemas distribuidos las vistas de implementación y despliegue pasan a un plano importante

3.2.8 Ciclo de vida del desarrollo de software
Metodología: Proceso Unificado de Rational (I)

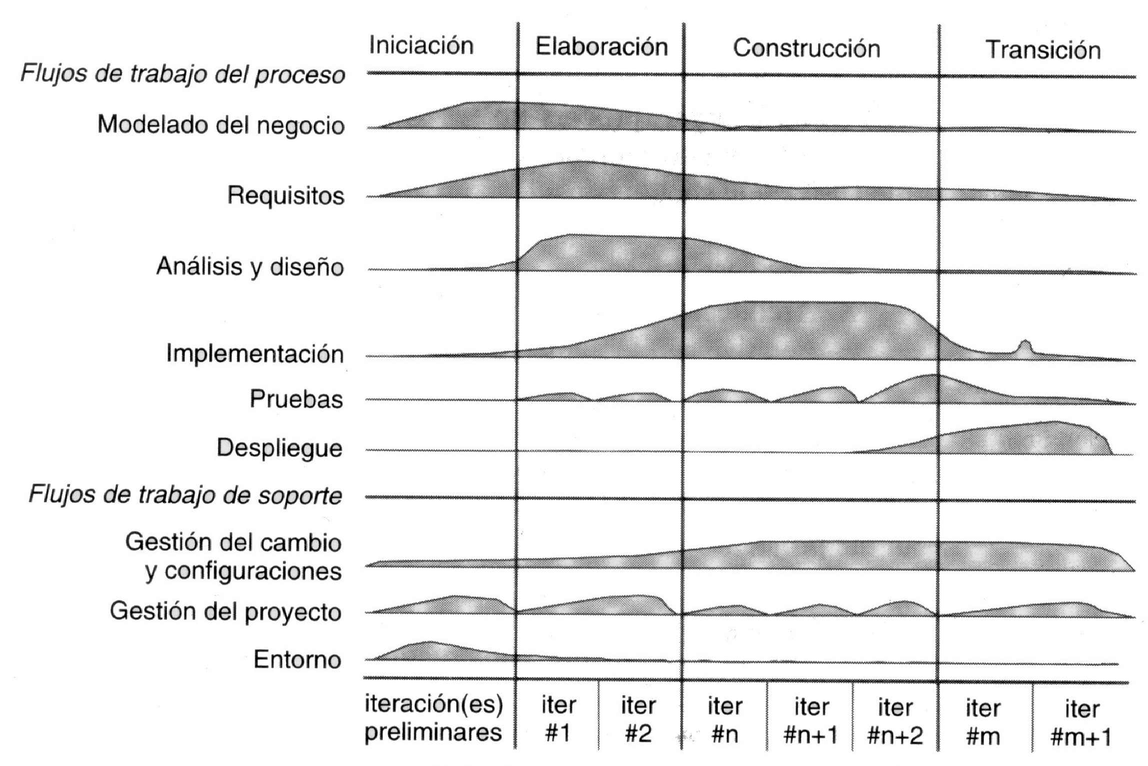
[Booch 99, capítulo 2, apendice C] [Jacobson 99]

- Se caracteriza por
  - **Está dirigida por los casos de uso**
    - Se utilizan para establecer el comportamiento deseado por el sistema, para verificar y validar la arquitectura del sistema, para las pruebas y para la comunicación entre las personas del proyecto
  - **Centrado en la arquitectura**
    - La arquitectura se utiliza para conceptualizar, construir, gestionar y hacer evolucionar el sistema en desarrollo
  - **Iterativo e incremental**
    - **Proceso iterativo** es aquél que involucra la gestión de un flujo de ejecutables
    - **Proceso incremental** es aquél que involucra la continua integración de la arquitectura del sistema para producir esos ejecutables, donde cada ejecutable incorpora mejoras incrementales sobre los otros.
    - Un proceso iterativo e incremental está *dirigido por el riesgo*, lo que significa que cada versión se encarga de atacar y reducir los riesgos más significativos para el éxito del proyecto

3.2.8 Ciclo de vida del desarrollo de software Metodología: Proceso Unificado de Rational (II)

[Booch 99, capítulo 2, apendice C] [Jacobson 99]

- Se descompone en fases
  - Una **fase** es el intervalo de tiempo entre dos hitos importantes del proceso, cuando se cumplen un conjunto de objetivos bien definidos, se completan los artefactos y se toman las decisiones sobre si pasar o no a la siguiente fase
- Hay cuatro fases
  - Iniciación
  - Elaboración
  - Construcción
  - Transición
- Una **iteración** es un conjunto bien definido de actividades, con un plan y unos criterios de evaluación bien establecidos, que acaba en una versión, bien interna o externa



Ciclo de vida del desarrollo de software.

3.2.9 Herramientas CASE

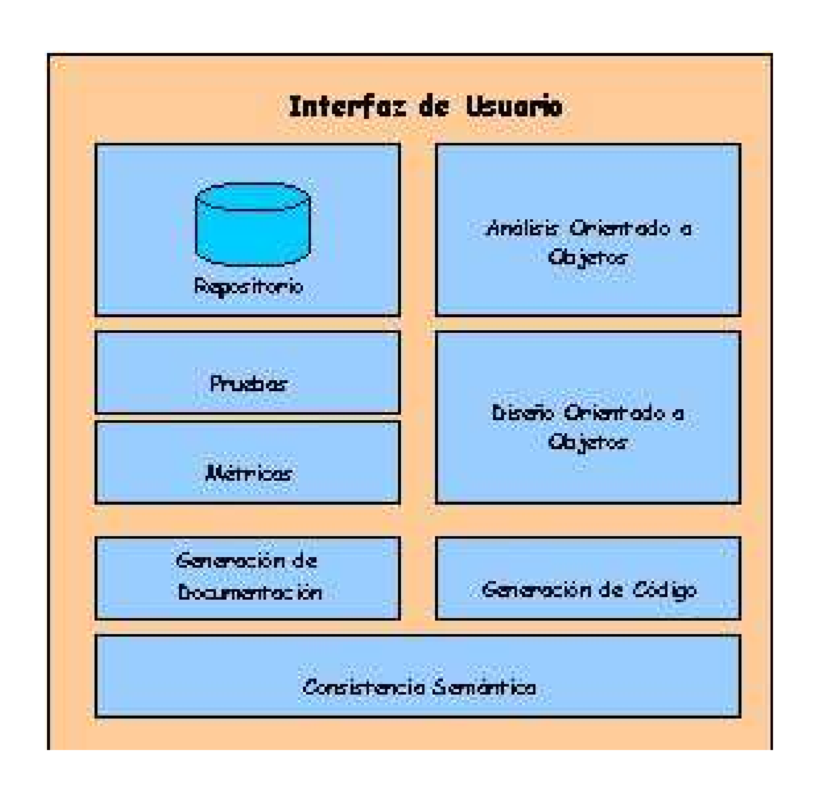
(Computer Aided/assisted Software/System Engineering)

[Piattini 1996]

- La tecnología CASE persigue la automatización del desarrollo de software
- Las herramientas CASE permiten mejorar la calidad y la productividad de los sistemas de información
  - Permitir la aplicación práctica de metodologías, lo que resulta muy difícil sin emplear herramientas.
  - Facilitar la realización de prototipos y el desarrollo conjunto de aplicaciones.
  - Simplificar el mantenimiento de los programas.
  - Mejorar y estandarizar la documentación.
  - Aumentar la portabilidad de las aplicaciones.
  - Facilitar la reutilización de componentes software.
  - Permitir un desarrollo y un refinamiento visual de las aplicaciones, mediante la utilización de gráficos

3.2.10 Elementos de una herramienta CASE

- ***Repositorio*** (diccionario) donde se almacenan los elementos definidos o creados por la herramienta, y cuya gestión se realiza mediante el apoyo de un SGBD o de un sistema de gestión de ficheros.
- ***Análisis Orientado a Objetos****,* es la parte de la herramienta donde se introducen la información relativa al Análisis Orientado a Objetos. Así pueden introducirse los documentos de análisis, requisitos, escenarios, actores, interfaces con usuarios, casos de uso, etc.
- ***Diseño Orientado a Objetos****,* es la parte de la herramienta donde se introduce la información relativa al diseño orientado a objetos. Así pueden introducirse los modelos lógico y físico del sistema. El objetivo fundamental es alimentar el diccionario de datos (o clases) del repositorio.
- ***Pruebas****,* que verifiquen los diferentes aspectos del proyecto.
- ***Métricas****,* es una asignación de un valor a un atributo (tiempo, complejidad, etc.) de una entidad software, ya sea un producto (código, diseño, etc.) o un proceso (pruebas, diseño, etc.)
- ***Generador de documentación***, que permite obtener la documentación que describe el sistema de información desarrollado; documentación que está asociada a las técnicas y notaciones empleadas.
- ***Generador de código****,* a partir de las especificaciones del diseño se puede generar código tanto para los programas como para la creación de los esquemas de bases de datos.
- ***Consistencia Semántica****,* facilidades que permiten llevar a cabo un análisis de la exactitud, integridad y consistencia de los esquemas generados por la herramienta.
- ***Interfaz de usuario***, que constará de editores de texto y herramientas de diseño gráfico que permitan, mediante la utilización de un sistema de ventanas, iconos y menús, con la ayuda del ratón, definir los diagramas, etc., que incluyen las diferentes notaciones.

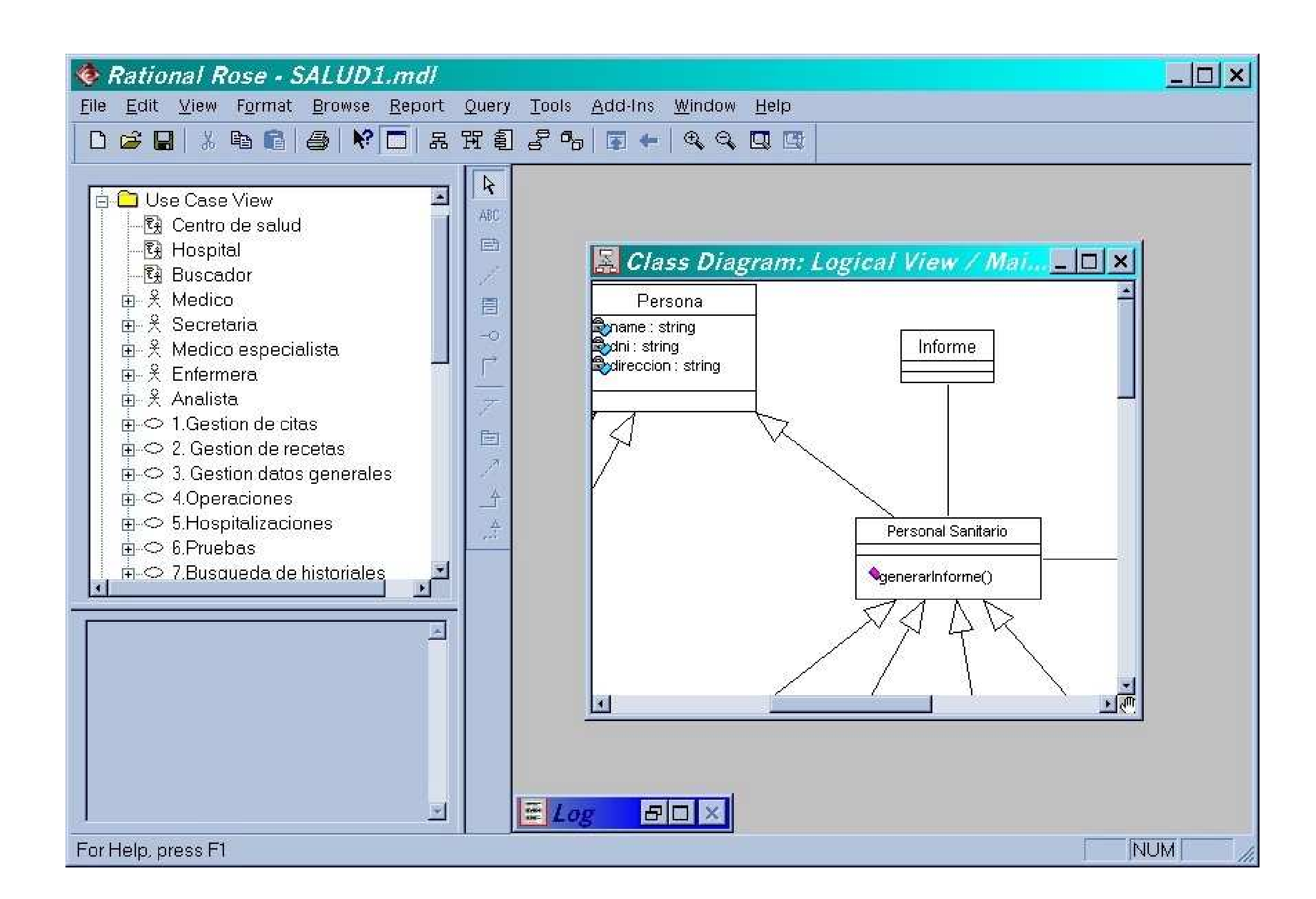


3.2.11 Criterios de selección de herramientas CASE

- Soporte completo de una notaciones
- Interfaz de usuario amigable
  - Consistente con la notación
  - Debe realizar comprobaciones mientras se está diseñando
    - Ejemplo: No debe permitir que una clase padre herede de una clase hija
- Generación de código automático
  - Normalmente generan declaraciones y algunos métodos básicos como constructores y destructores
- Ingeniería inversa a partir de código legado o código generado
  - A partir de ficheros con programas se obtiene los esquemas de la notación
  - Cambios en el código implican cambios en la notación
- Seguimiento de los requisitos
  - Reflejar en el diseño y en el código los requisitos
- Generación automática de la documentación de la aplicación
  - La documentación producida debe ser
    - Completa
    - Bien diseñada, cómoda de utilizar y de usar
    - Personalizable (se genera en formatos estándar que pueden ser manejados por herramientas de maquetación estándar)
- Comprobación de la consistencia y completud entre todas las vistas del sistema
- Ayuda en línea completa
- Buena documentación de la herramienta

3.2.12 Herramienta Rational Rose ®

- Permite el uso de la notación UML
- Genera varios lenguajes: Java, C++, Visual Basic, ...
- Soporta el desarrollo basado en componentes
- Permite el trabajo en equipo

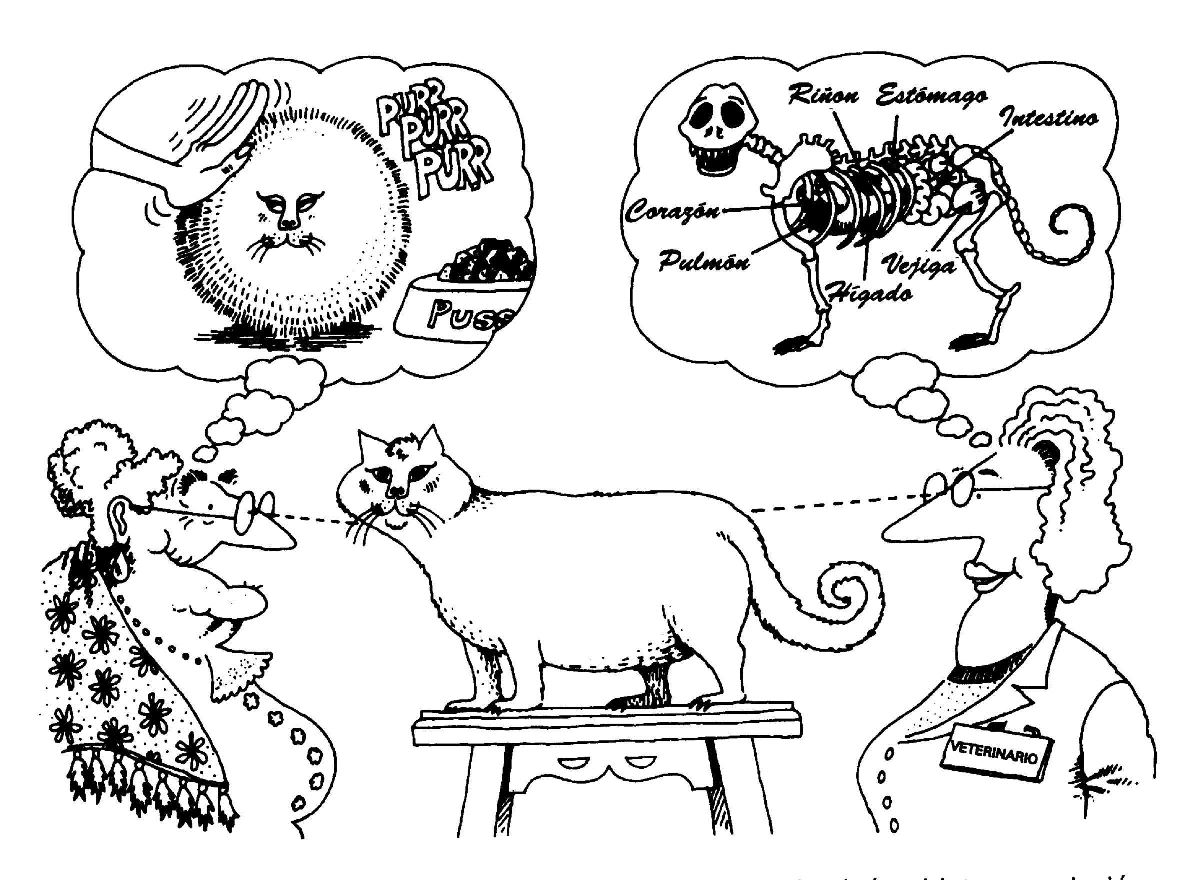


3.3 Elementos del modelo de objetos

- **Elementos fundamentales**
  - Abstracción
  - Encapsulación y ocultación de la información
  - Modularidad
  - Jerarquía (herencia, polimorfismo y agregación)
- **Elementos secundarios**
  - Tipos (control de tipos)
  - Concurrencia
  - Persistencia
  - Distribución
  - Sobrecarga
  - Genericidad
  - Manejo de excepciones
- **Elementos relacionados**
  - Componentes
  - Patrones de diseño

3.4 Abstracción

Una abstracción denota las características esenciales de un objeto que lo distinguen de todos los demás tipos de objeto y proporciona así fronteras conceptuales nítidamente definidas respecto a la perspectiva del observador [Booch 94]



La abstracción se centra en las características esenciales de algún objeto, en relación a la perspectiva del observador.

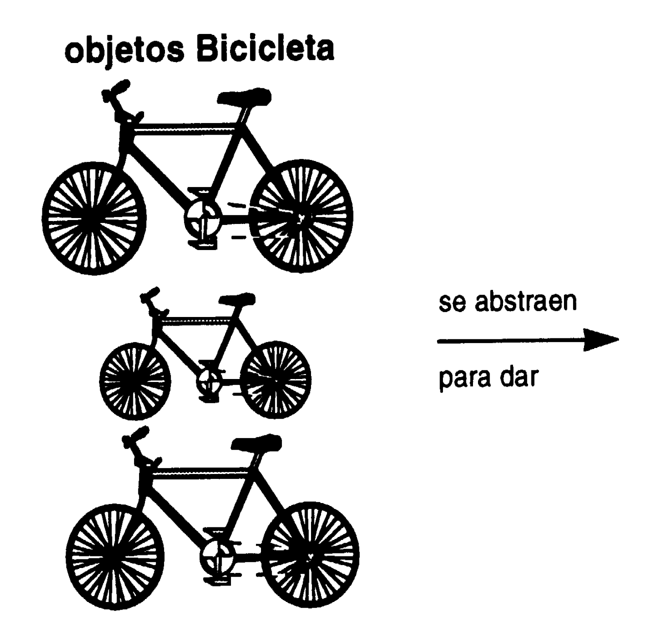
Abstracción

- En la abstracción obtenemos objetos del mundo real de los cuales enfatizaremos algunos detalles o propiedades mientras suprimiremos otros
- También nos fijaremos en el comportamiento de los objetos para definir las acciones u operaciones que son capaces de realizar
- Los objetos en el sistema se deben de tratar como seres vivos que nacen (constructores), viven (están activos), duermen (están pasivos), se reproducen (copia de objetos), y mueren por eutanasia (destructores) o por inanición (recolector de basura)
- Ejemplo de objetos: una factura, un albarán, un usuario, ...
- A partir de objetos que tienen unas propiedades y acciones comunes se deben abstraer las clases. Los nombres de los objetos son nombres propios.
- En las clases se definen las propiedades y operaciones comunes de los objetos. Los nombres de las clases son nombres comunes.
- Los nombres de los métodos son verbos, pues especifican las acciones que realizan los objetos



Clases y objetos

[Rumbaught 91]



clase Bicicleta

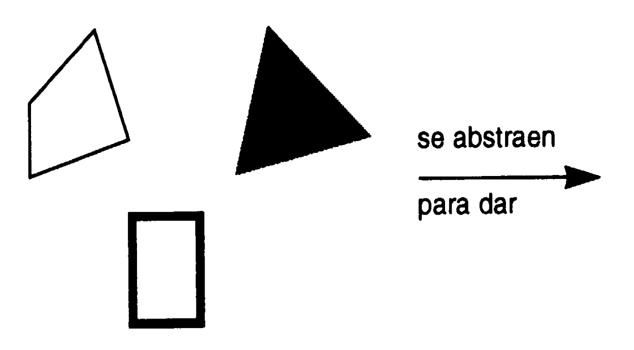
Atributos

tamaño del cuadro
tamaño de rueda
marchas
material

Operaciones

cambiar marcha
mover
reparar

objetos Polígono



clase Polígono

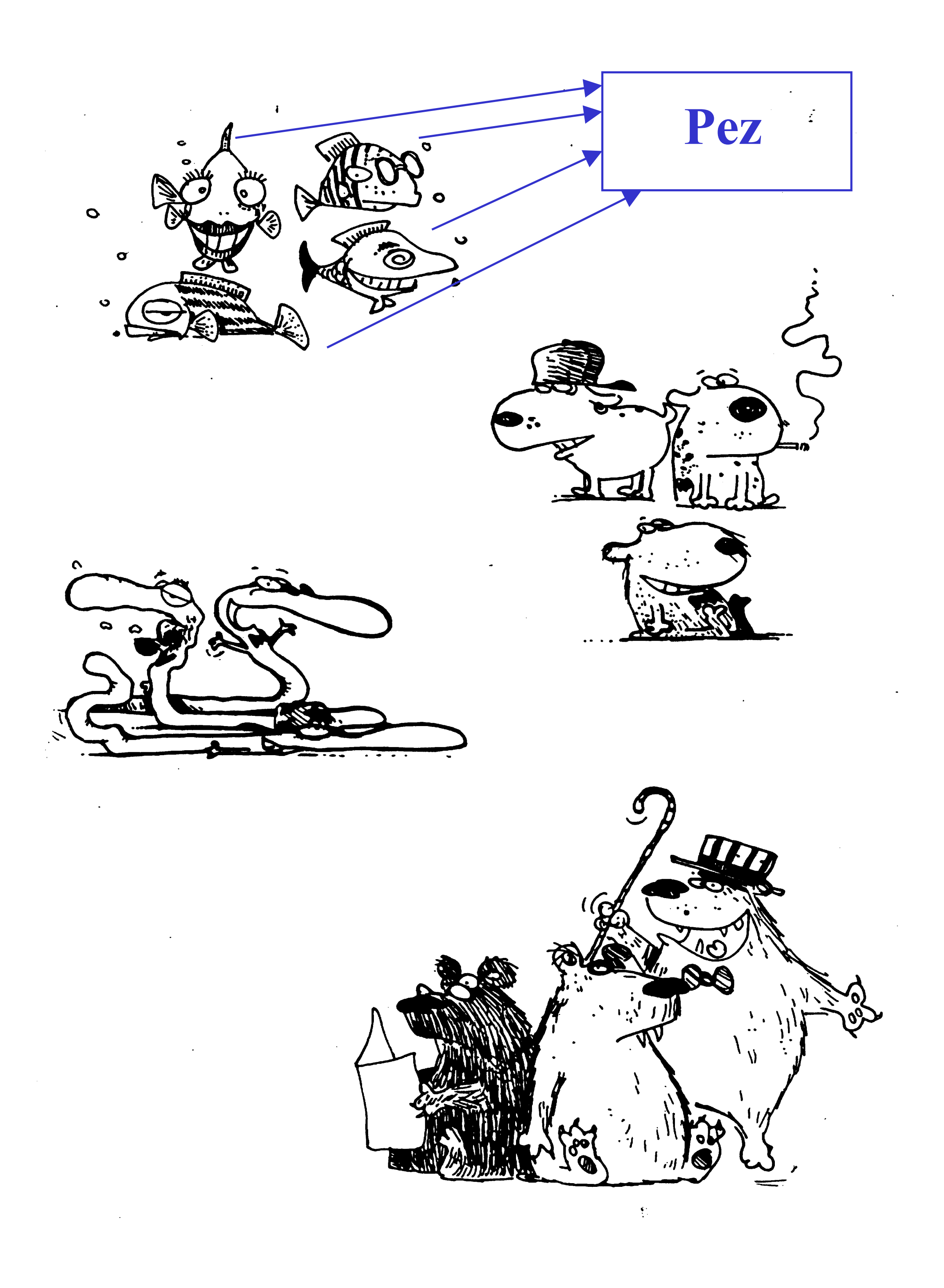
Atributos

vértices
color del borde
color del interior

Operaciones

dibujar
borrar
mover

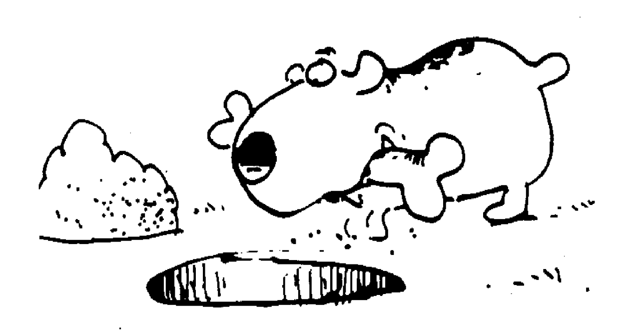
Ejemplos de objetos de las clases Pez, Perro, Serpiente y Oso



Comportamiento común de los objetos de la clase *Perro*

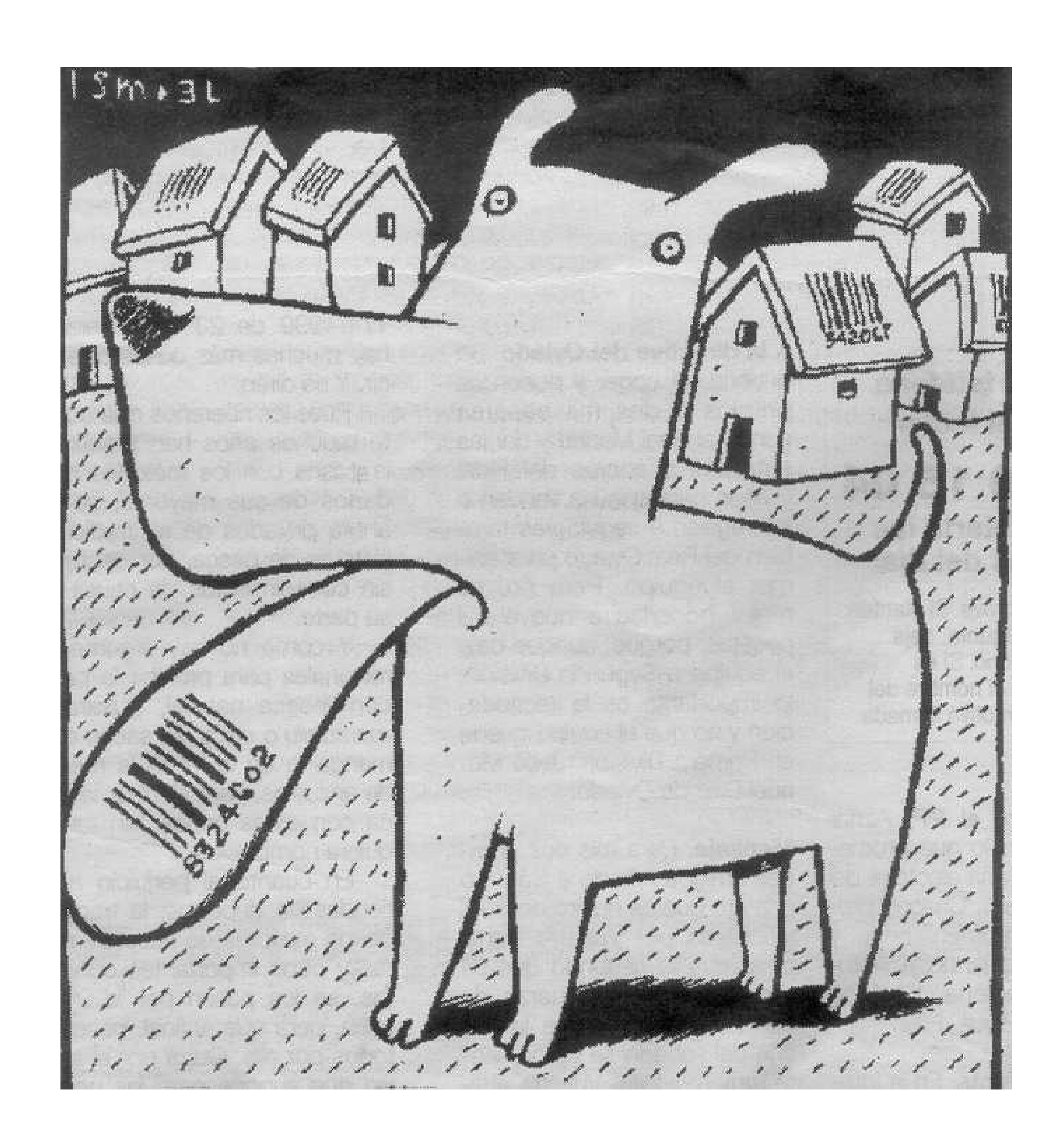
Perro
nombre: string # raza: string # nacimiento: fecha # identificacion: long
+ marcar_territorio (x,y, z) + guardar (hueso) + recuperar (hueso)





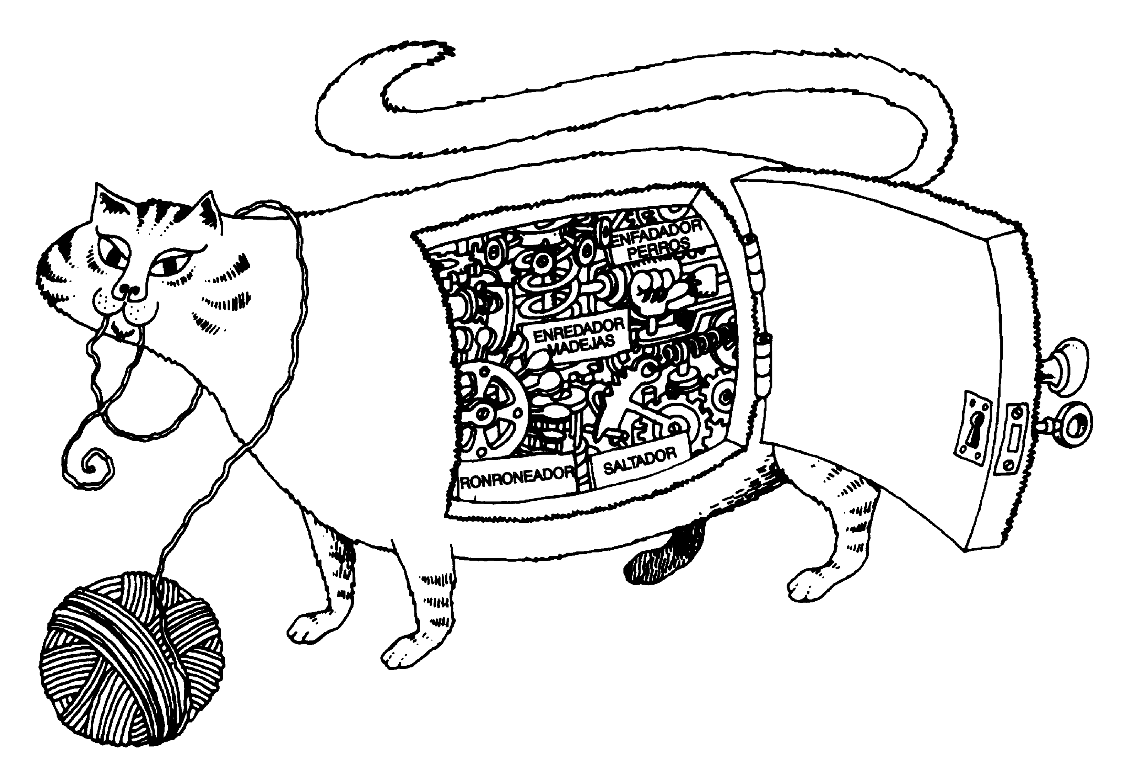
Identidad de objetos

En las clases deben diseñarse atributos de forma que permitan identificar a los objetos de manera única



3.5 Encapsulación

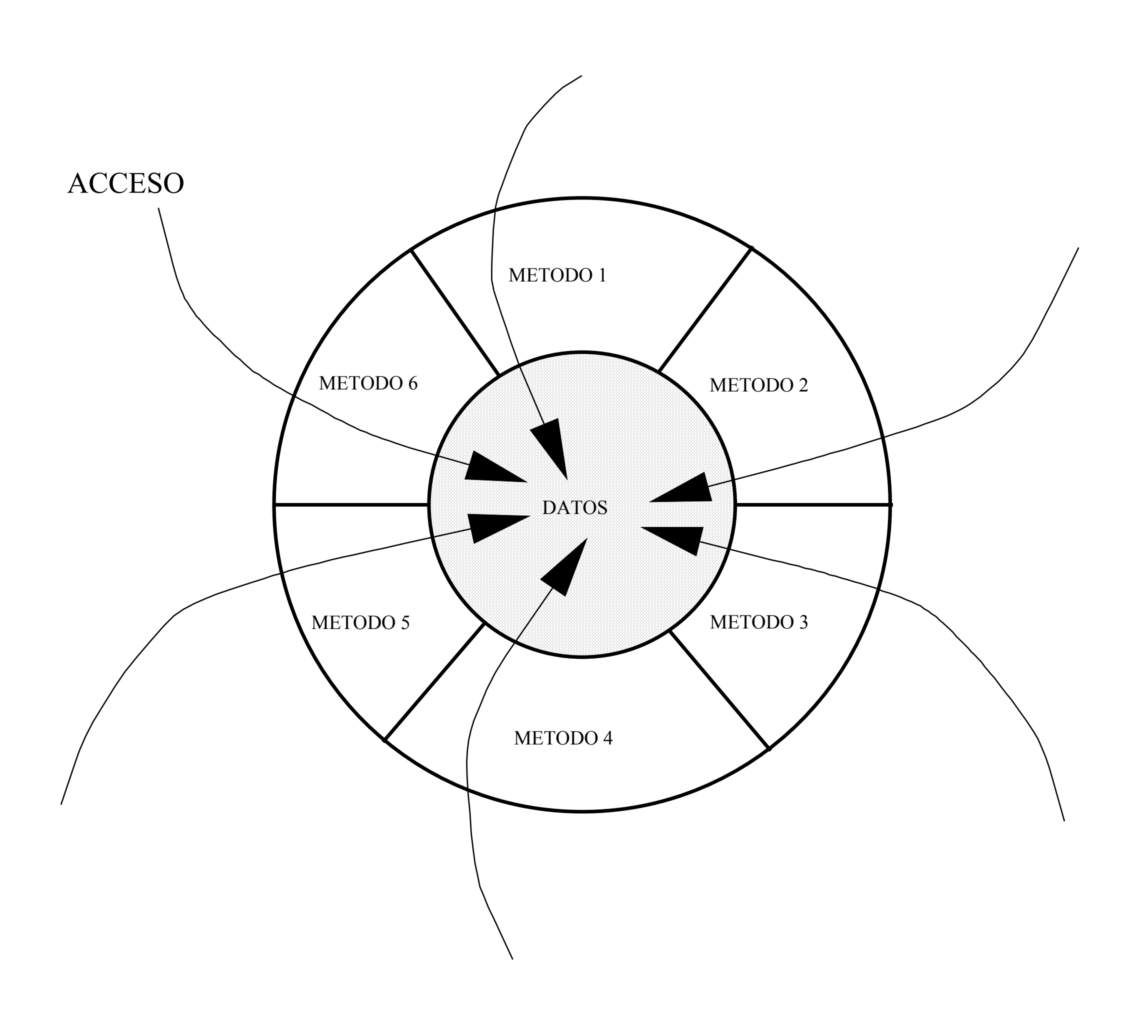
Es el proceso de almacenar en un mismo compartimento los elementos de una abstracción que constituyen su estructura y su comportamiento; sirve para separar el interfaz contractual de una abstracción y su implementación [Booch 94]



El encapsulamiento oculta los detalles de la implementación de un objeto.

Encapsulación

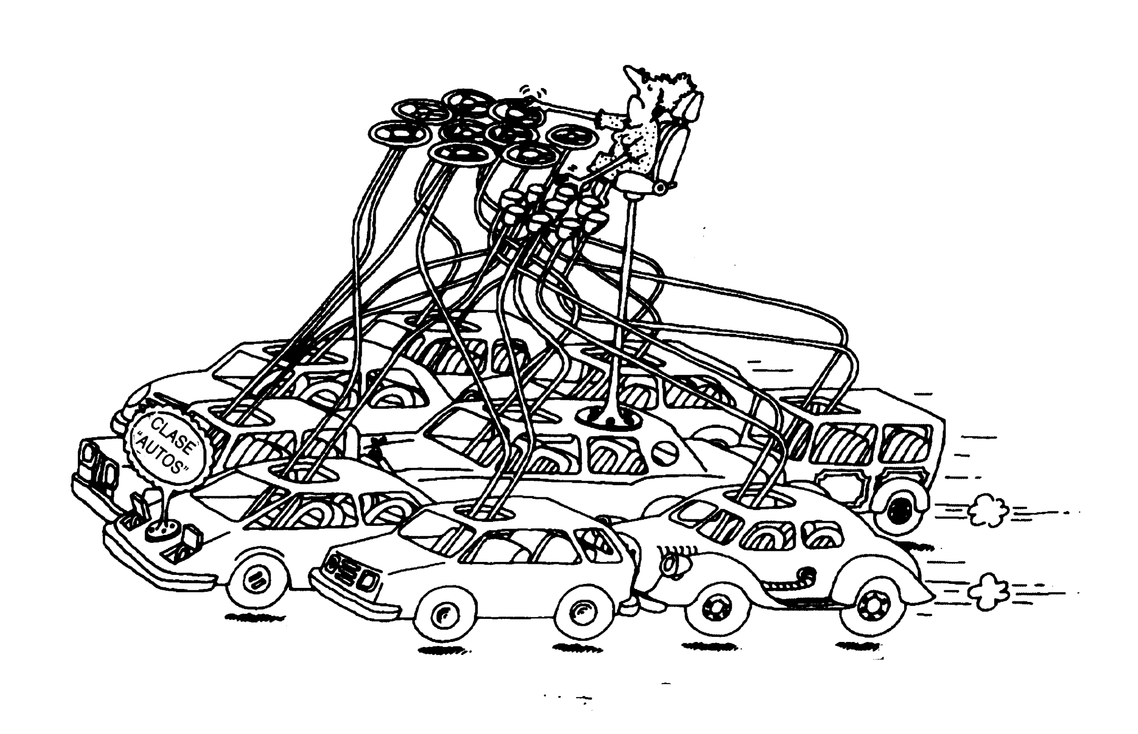
En la encapsulación los datos o propiedades son privados y se accede a ellos a través de los métodos [Cueva 93]



Clases

Una clase es un conjunto de objetos que comparten una estructura común y un comportamiento común

[Booch 94]



Una clase representa un conjunto de objetos que comparten una estructura común y un comportamiento común.

¿Cómo se modela la señora en la clase autos?

¿Cómo todos los objetos de una clase comparten algo?

Clases

- Los atributos y las operaciones pueden ser visibles o no según el detalle deseado
- Un atributo es un dato que se almacenará en los objetos que son instancias de la clase.
- Un método es la implementación de una operación para la clase
- Un adorno de clase es una propiedad por ejemplo indicar que la clase es abstracta (no puede tener instancias, sólo se puede heredar de ella)
- En UML las clases se pueden representar de tres formas:
  – Sin detalle
  – Detalles a nivel de análisis y diseño
  – Detalle a nivel de implementación
- Los atributos en UML se pueden describir como:

  visibilidad : *tipo* = *valor-inicial* { *propiedad* }
  - donde *visibilidad* puede ser:
    – + publico
    – # protegido
    – - privado

UML

Window

Window
size: Area
visibility: Boolean
display ()
hide ()

Window
{abstract,
author=Joe,
status=tested}
+size: Area = (100,100)
#visibility: Boolean = invisible
+default-size: Rectangle
#maximum-size: Rectangle
-xptr: XWindow*
+display ()
+hide ()
+create ()
-attachXWindow(xwin:Xwindow*)

Diseño de la clase *Perro*

Perro
nombre: string="Sin nombre" # raza: string="Sin raza" # contador: long {static} # codigo: long
<<constructores>> + Perro(void) + Perro(string unNombre, string unaRaza) <<destructores>> + ~Perro(void) <<selectores>> + verNombre(void):string + verRaza(void):string + verCodigo(void):long + verContador(void):long {static} <<modificadores>> + void ponNombre(string unNombre); + void ponRaza(string unaRaza); <<comportamiento>> + void ladra(void); + void saluda(void);



Diseño de la clase *Perro* con Rational Rose ®

www.rational.com

Perro

codigo : Long = 0
contador : Long = 0
nombre : String = "Sin nombre"
raza : String = "Sin raza"

<<Destructor>> ~Perro()
<<Comportamiento>> ladra()
<<Constructor>> Perro()
<<Modificador>> ponNombre()
<<Modificador>> ponRaza()
<<Comportamiento>> saluda()
<<Selector>> verCodigo()
<<Selector>> verContador()
<<Selector>> verNombre()
<<Selector>> verRaza()

Implementación de la clase *Perro* en C++

Perro.hpp [Stroustrup 2000, capítulo 10]

```
// Clase Perro (archivo cabecera)
// versión 1.0. 12-Noviembre-2000
// autor: J.M. Cueva Lovelle

#ifndef PERRO_HPP
#define PERRO_HPP

#include <iostream>
#include <string>      //Para manejo de string
using namespace std;  //Para usar la biblioteca estandar

class Perro
{
 protected:
 string nombre;
 string raza;
 static long contador; //sólo pertenece a la clase
 long codigo;
 public:
 //constructores
 Perro (void);
 Perro (string unNombre, string unaRaza);
 //comportamiento
 void ladra (void) {cout<<nombre<<" dice guau, guau"<<endl;};
 void saluda(void);
 //selectores
 string verNombre(void) const { return nombre; };
 string verRaza(void) const { return raza; };
 long verCodigo(void) const { return codigo; };
 static long verContador(void) { return contador; };
 //modificadores
 void ponNombre(string unNombre) {nombre=unNombre; };
 void ponRaza(string unaRaza) {raza=unaRaza; };
 //destructor
 ~Perro(void) {cout<<"Se ha muerto el perro de nombre "<<nombre;
               cout<<" y codigo "<<codigo<<endl; };
};

#endif  //fin PERRO_HPP
```

Implementación de la clase *Perro* en C++

Perro.cpp [Stroustrup 2000, capítulo 10]

```
// Clase Perro (archivo de implementación)
// versión 1.0. 12-Noviembre-2000
// autor: J.M. Cueva Lovelle


#include "Perro.hpp"

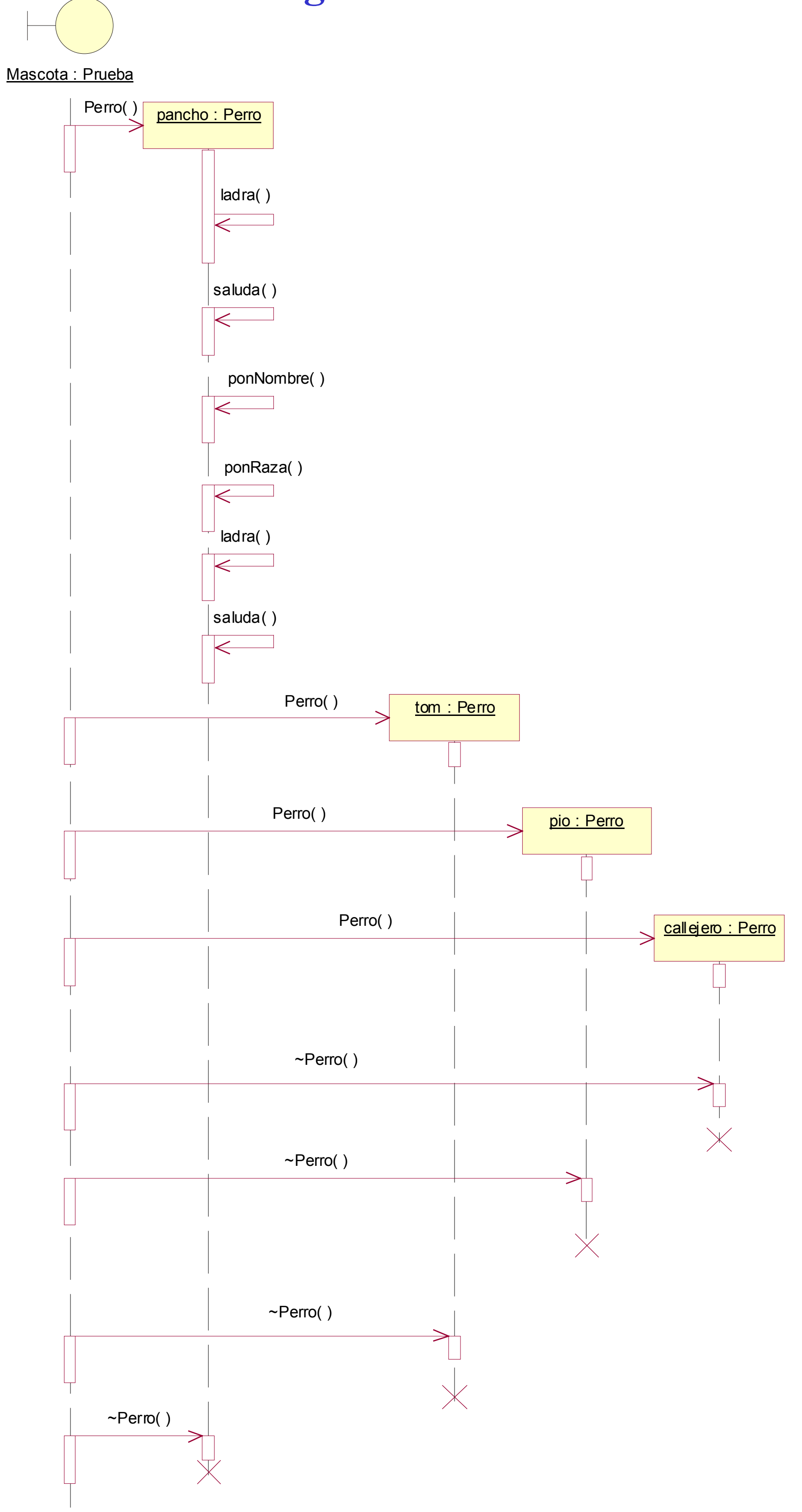
long Perro::contador=0;   //miembro estático, sólo pertenece a la clase

Perro::Perro(void)   // Constructor de la clase perro
{
 nombre="Sin nombre";
 raza="Sin raza";
 codigo=contador++;
 cout << "Ha nacido un perro que se llama: "<<verNombre()<<endl;
}

Perro::Perro(string unNombre, string unaRaza)
{                          // Constructor de la clase perro sobrecargado
 nombre=unNombre;
 raza=unaRaza;
 codigo=contador++;
 cout << "Ha nacido un perro que se llama: "<<verNombre()<<endl;
}

void Perro::saluda(void)
{
 cout<<"Hola, mi nombre es "<<verNombre()<<endl;
 cout<<"y soy de la raza "<<verRaza()<<endl;
 cout<<"Mi código interno es "<<verCodigo()<<endl;
}
```

Prueba unitaria de la clase *Perro* en UML
Diagrama de Secuencia



Prueba unitaria de la clase *Perro* en C++

Mascota.cpp [Stroustrup 2000, capítulo 10]

```
// Prueba unitaria de la Clase Perro
// versión 1.0. 12-Noviembre-2000
// autor: J.M. Cueva Lovelle
#include "Perro.hpp"
int main()
{
  Perro pancho; //Se llama al constructor por defecto
  pancho.ladra();
  pancho.saluda();

  pancho.ponNombre("Pancho");
  pancho.ponRaza("Setter");
  pancho.ladra();
  pancho.saluda();

  Perro tom ("tom", "pointer"); //Se llama al constructor
  tom.ladra();
  tom.saluda();

  Perro pio("pio","vulgaris"); //Se llama al constructor
  pio.ladra();
  pio.saluda();

  string unNombre, unaRaza;
  cout << "Dame el nombre de un perro: ";
  cin >> unNombre;
  cout << "y ahora su raza: ";
  cin >> unaRaza;
  Perro callejero(unNombre,unaRaza); //Se llama al constructor
  callejero.ladra();
  callejero.saluda();

  cout << "El numero de perros creados por la clase Perro es: ";
      //Al ser un método estático se aplica sobre la clase
  cout << Perro::verContador() << endl;

  return 0;
} //Se llama al destructor al finalizar el bloque
```

Ejecución de la prueba unitaria de la clase *Perro* en C++

```
Ha nacido un perro que se llama: Sin nombre
Sin nombre dice guau, guau
Hola, mi nombre es Sin nombre
y soy de la raza Sin raza
Mi código interno es 0
Pancho dice guau, guau
Hola, mi nombre es Pancho
y soy de la raza Setter
Mi código interno es 0
Ha nacido un perro que se llama: tom
tom dice guau, guau
Hola, mi nombre es tom
y soy de la raza pointer
Mi código interno es 1
Ha nacido un perro que se llama: pio
pio dice guau, guau
Hola, mi nombre es pio
y soy de la raza vulgaris
Mi código interno es 2
Dame el nombre de un perro: Ron
y ahora su raza: mastin
Ha nacido un perro que se llama: Ron
Ron dice guau, guau
Hola, mi nombre es Ron
y soy de la raza mastin
Mi código interno es 3
El numero de perros creados por la clase Perro es: 4
Se ha muerto el perro de nombre Ron y codigo 3
Se ha muerto el perro de nombre pio y codigo 2
Se ha muerto el perro de nombre tom y codigo 1
Se ha muerto el perro de nombre Pancho y codigo 0
```

• Para compilar con el compilador de GNU (**www.gnu.org**) en linux

`$ g++ -o Mascota.out Mascota.cpp Perro.cpp`

• Para ejecutar el programa

`$./Mascota.out`

Prueba unitaria de la clase *Perro* en C++ Utilizando C++ Builder (Consola)

Mascota.cpp (I)

```
// Prueba unitaria de la Clase Perro
// versión 1.0. 12-Noviembre-2000
// autor: J.M. Cueva Lovelle
// Compilado con C++ Builder Versión 4.0
#include "Perro.hpp"
//---------------------------------------------------------
#pragma hdrstop
#include <condefs.h>
//---------------------------------------------------------
// Para el uso de getch() se incluye conio
#include<conio>
//---------------------------------------------------------
USEUNIT("Perro.cpp");
//---------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
  Perro pancho; //Se llama al constructor por defecto
  pancho.ladra();
  pancho.saluda();

  pancho.ponNombre("Pancho");
  pancho.ponRaza("Setter");
  pancho.ladra();
  pancho.saluda();

  Perro tom ("tom", "pointer"); //Se llama al constructor
  tom.ladra();
  tom.saluda();
```

Prueba unitaria de la clase *Perro* en C++ Utilizando C++ Builder (Consola)
Mascota.cpp (II)

```
  Perro pio("pio","vulgaris"); //Se llama al constructor
  pio.ladra();
  pio.saluda();

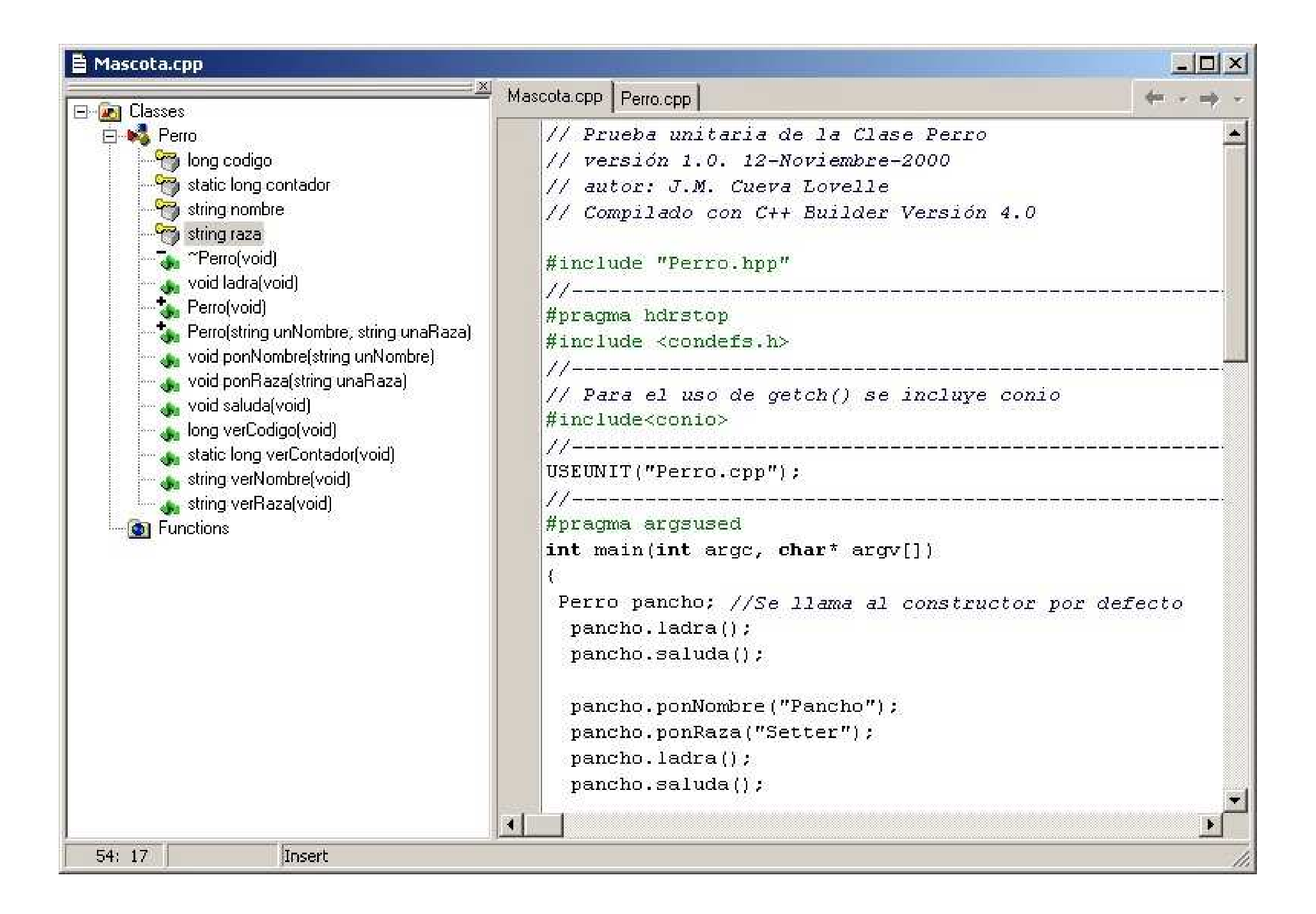
  string unNombre, unaRaza;
  cout << "Dame el nombre de un perro: ";
  cin >> unNombre;
  cout << "y ahora su raza: ";
  cin >> unaRaza;
  Perro callejero(unNombre,unaRaza); //Se llama al constructor
  callejero.ladra();
  callejero.saluda();

  cout << "El numero de perros creados por la clase Perro es: ";
      //Al ser un método estático se aplica sobre la clase
  cout << Perro::verContador() << endl;

  cout<<"Pulse una tecla para finalizar programa" ;
  getch(); //Espera que se pulse una tecla

  return 0;
}
```

Prueba unitaria de la clase *Perro* en C++
Utilizando C++ Builder (Consola)
Entorno de desarrollo



Conclusiones para el diseño de clases

- Una ***clase*** es un tipo definido por el usuario, que permite crear objetos. Las clases tienen miembros formados por campos y métodos. Una *struct* es una clase con todos los miembros públicos.
- Los campos miembro suelen declararse protegidos (***protected***), salvo en casos especiales que se declaran privados (***privated***)
- Los métodos públicos (***public***) permiten acceder a los campos miembro desde el exterior de la clase
- Los ***métodos selectores*** no modifican el valor de los campos miembro. Se suelen denominar:
  - ***ver...campoMiembro( )*** (en castellano). ¡ ver, pero no tocar !
  - ***get...campoMiembro( )*** (en inglés).
  - Suele declararse un método por cada tipo de campo miembro accesible
  - Son una garantía para que no se modifiquen accidentalmente los campos miembro
- Los ***métodos modificadores*** permiten modificar los campos miembro de la clase. Se suelen denominar
  - ***pon...campoMiembro( )*** (en castellano).
  - ***set...campoMiembro( )*** (en inglés).
  - Suele declararse un método por cada tipo de campo miembro accesible
  - Establecen el camino de cómo se modifican los campos miembro y realizan las comprobaciones necesarias.
- Los ***métodos iteradores*** permiten recorrer las partes de un objeto
- Los ***constructores*** crean los objetos
  - Pueden estar *sobrecargados (overload)*. Se pueden definir varios métodos con el mismo identificador pero que difieran en número o tipo de parámetros.
  - Se aconseja poner siempre constructores, para saber donde y cuando se crean los objetos.
  - Se puede poner un mensaje que indique cuando y donde se están construyendo los objetos. Muy aconsejable para depurar programas.
- Los ***destructores*** se encargan de eliminar los objetos
  - Se aconseja poner siempre destructores (en C++), para saber donde y cuando se destruyen los objetos.
  - Se puede poner un mensaje que indique cuando y donde se están destruyendo los objetos. Muy aconsejable para depurar programas

Diseño de la clase *Vehiculo*

Vehiculo

matricula: string="Sin matricular"
propietario: string="Sin propietario"
velocidadMaxima: float
numeroPasajeros: int
numeroVehiculosProducidos: long {static}
numeroSerie: long

<<constructor>>
+ Vehiculo()
+ Vehiculo(unaMatricula:String, unPropietario:String,unaVelocidad:float, unNumeroPasajeros:int)

<<selector>>
+ verMatricula():String
+ verPropietario():String
+ verVelocidadMaxima():float
+ verNumeroPasajeros():int
+ verNumeroSerie():long
+ verVehiculosProducidos():long {static}

<<utilidad>>
+ toString():String

Implementación de la clase *Vehiculo* en Java
Declaración de campos

```
/**
 * clase Vehiculo, ilustra el manejo básico de clases
 * @version 1.0 15 de Julio 1998
 * @author Juan Manuel Cueva Lovelle
 */

// Solo los identificadores de clases pueden empezar con mayúsculas

class Vehiculo {

   // campos
   //declaración e inicialización de cadenas
   protected String matricula= "Sin matricular” ;
   protected String propietario= "Sin propietario” ;

   // Los campos numéricos se inializan implícitamente
   protected float velocidadMaxima; // inicializado a cero
   protected int numeroPasajeros; //inicializado a cero

   // Campos estáticos
   //Son campos que se comparten por objetos de la clase
   //Son definidos en la clase por eso también se llaman variables de clase
   protected static long numeroVehiculosProducidos;

   private long numeroSerie; //almacena en cada objeto el número de serie
```

Implementación de la clase *Vehiculo* en Java
Implementación de constructores

```
// Constructores
// No pueden devolver nada pero pueden tener parámetros
// Si no se definen siempre existe un constructor por defecto (no-arg)

//Constructor mínimo
//También podría tener parámetros

Vehiculo(){
  numeroSerie=numeroVehiculosProducidos++;
  }

//Sobrecarga de constructores
//Constructor completo
Vehiculo(String unaMatricula,
         String unPropietario,
         float unaVelocidadMaxima,
         int unNumeroPasajeros){

         this(); //Invocación de constructor explícito
                 //Debe ser la primera instrucción ejecutable
                 //También puede tener parámetros
                 //La sobrecarga se resuelve en tiempo de compilación
         matricula = unaMatricula;
         propietario = unPropietario;
         velocidadMaxima = unaVelocidadMaxima;
         numeroPasajeros = unNumeroPasajeros;
         }
```

Implementación de la clase *Vehiculo* en Java
Implementación de métodos selectores

```
//Métodos. Para Java los constructores no son métodos
    // Java no acepta métodos con un número variable de parámetros como en C++
    // Tampoco tiene métodos inline como en C++
    // Los parámetros de los métodos se pasan por valor

    public String verMatricula(){
      return matricula; // métodos públicos para acceder a campos protegidos
      }

    public String verPropietario(){
      return propietario;
      }

    public float verVelocidadMaxima(){
      return velocidadMaxima;
      }

    public int verNumeroPasajeros(){
      return numeroPasajeros;
      }

    public long verNumeroSerie(){
      return numeroSerie;
      }

// Método estático
    //Sólo para la clase
    //Sólo puede manejar campos estáticos y métodos estáticos
    //Para utilizarlo se debe usar el nombre de la clase
    //     Vehiculo.verVehiculosProducidos

    public static long verVehiculosProducidos(){
      return numeroVehiculosProducidos;
      }

    public void verTodo(){
       System.out.println("Matrícula: " + verMatricula()+'\n'+
                          "Nombre del propietario: ” +verPropietario()+'\n'+
                          "Velocidad máxima: ” +verVelocidadMaxima()+ '\n'+
                          "Número de pasajeros: ” +verNumeroPasajeros()+'\n'+
                          "Número de serie: ” + verNumeroSerie());
                          }
```

Implementación de la clase *Vehiculo* en Java

Implementación del método *toString*

```
// Método toString
     // Es un método especial
     // Si un objeto tiene un método toString sin parámetros y que devuelve
     // un String, entonces este método será invocado si aparece el identificador
     // del objeto en una concatenación de cadenas String con el operador
     // concatenación +

     public String toString(){
       String descriptor = "Número de Serie -” + verNumeroSerie()+
                         "(” + verMatricula()+ ")” ;
       return descriptor;
       }
```

Implementación de la clase *Vehiculo* en Java
Implementación del método *main*
Prueba Unitaria de la clase

```java
/* Método main
        + Una aplicación puede tener cualquier número de métodos main,
          puesto que cada clase puede tener uno
        + Se puede usar el método main para la prueba unitaria de cada
          clase
        + El método main debe ser:
                    - public: Accesible exteriormente
                    - static: ligado a la clase no a los objetos
                    - void: no devuelve nada
                    - String []: Sólo puede aceptar un argumento String[],
                                 que son los argumentos del programa desde
                                 el sistema operativo
     */

     public static void main(String[] args){

      // Se crean dos objetos con el operador new

      Vehiculo cocheNuevo= new Vehiculo(); //Creación del objeto cocheNuevo
      Vehiculo renault11= new Vehiculo( "O-6297-AG" , "JMCL" ,150,5);

      cocheNuevo.verTodo();
      renault11.verTodo();


      // Prueba de métodos estáticos
      System.out.println( "Número de vehículos producidos: " +
                          Vehiculo.verVehiculosProducidos());

      // Prueba de toString
      System.out.println( "Objeto cocheNuevo con toString: " + cocheNuevo);
      System.out.println( "Objeto renault11 con toString: " + renault11);

      // Creación de otro objeto
      Vehiculo otroCoche;
      // Aqui otroCoche es una referencia que contiene null

      otroCoche= new Vehiculo("O-5440-AB" ,"JMCL" ,140,5);
      // con new se ha reservado memoria en el área heap

      otroCoche.verTodo();
      System.out.println( "Número de vehículos producidos: " +
                          Vehiculo.verVehiculosProducidos());

      } // fin de main
 } // fin de la clase Vehiculo
```

Implementación de la clase Perro en C# (I)

```
using System;

namespace Mascota
{
    /// <summary>
    /// Modelado básico de la clase Perro
    /// </summary>
    class Perro
    {
            protected string nombre="Sin nombre";
            //Propiedad Nombre
            public string Nombre
            {
                        set {nombre=value;}
                        get {return nombre;}
            }

            protected string raza="Sin raza";
            //Propiedad Raza
            public string Raza
            {
                        set {raza=value;}
                        get {return raza;}
            }

            protected int codigo;
            protected static int contador=0;

            // Propiedad Edad
            protected int edad=0;

            public int Edad
            {
                        set
                        {
                         if (value>=0)
                                    edad=value;
                         else
                                    Console.WriteLine("Edad no válida");
                        }
                        get{return edad;}
            }
```

Implementación de la clase Perro en C# (II)

```
//Sobrecarga de constructores
    public Perro()
            {
                        codigo=contador++;
                        Console.WriteLine("Ha nacido el perro "+nombre);
            }

    public Perro(string nombre)
            {
                        this.nombre=nombre;
                        codigo=contador++;
                        Console.WriteLine("Ha nacido el perro "+nombre);
            }

    public Perro(string nombre, string raza)
            {
                        this.nombre=nombre;
                        this.raza=raza;
                        codigo=contador++;
                        Console.WriteLine("Ha nacido el perro "+nombre+
                                          " y de raza "+raza);
            }

//Implementación de un método estático
    public static int VerContador()
            {
                        return contador;
            }

    public void Ladra()
            {
                        Console.WriteLine(nombre+" dice guau ...");
            }

//Redefinición del método ToString() de Object
    public override string ToString()
            {
                        return ("Perro:"+nombre+" Raza:"+raza+"
                                 Codigo:"+codigo+" Edad:"+edad);
            }
```

Implementación de la clase Perro en C# (III)

```
/// <summary>
/// El método Main contiene la prueba unitaria de la clase perro
/// </summary>
     [STAThread]
     static void Main(string[] args)
     {
      Perro callejero = new Perro();
       Perro pancho=new Perro("Pancho");
       Perro tom= new Perro("Tom","Pointer");

       callejero.Nombre="Ron";
       callejero.Raza="Mastín";
       callejero.Edad=2;
       callejero.Ladra();

       Console.WriteLine("Dime lo que sabes de ... "+callejero);
       Console.WriteLine("Dime lo que sabes de ... "+pancho);
       Console.WriteLine("Dime lo que sabes de ... "+tom);

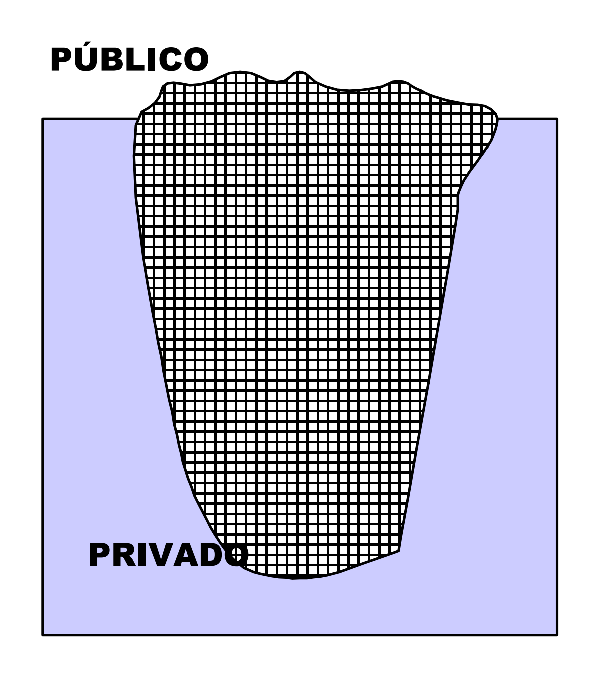
//Ejemplo de llamada a método estático
       Console.WriteLine("Los perros creados son "+Perro.VerContador());

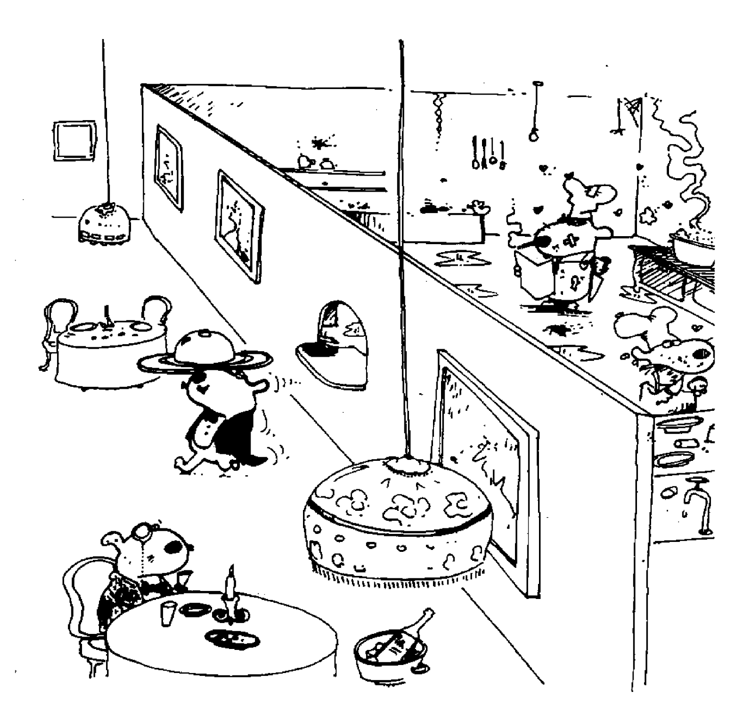
//Espera una tecla para finalizar
       Console.ReadLine();

       } //Fin de Main
     } // Fin de la clase Perro
} // Fin del namespace Mascota
```

Ocultación de información

Debe ser posible en una clase especificar que características están disponibles a todos los clientes, a los herederos, a algunos clientes o sólo se pueden utilizar internamente





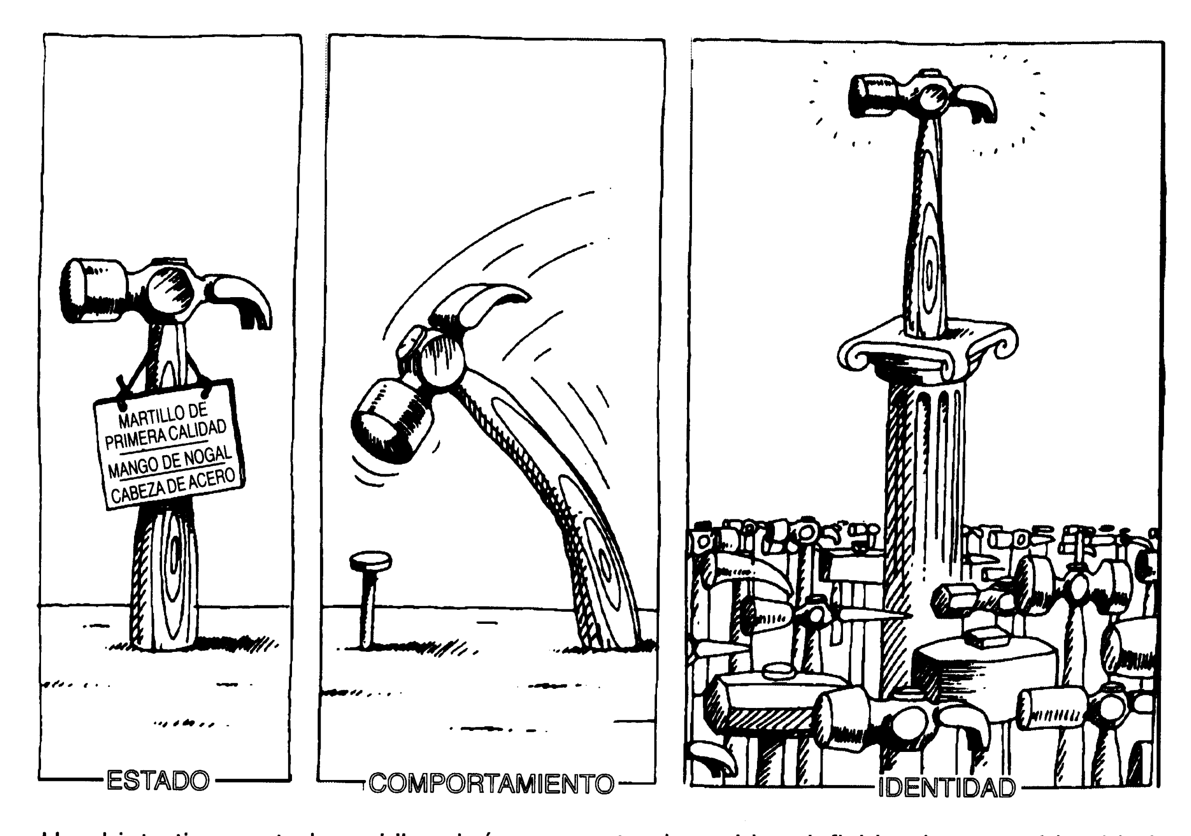
Encapsulación y ocultación de información

- Con la encapsulación los usuarios de una clase no necesitan conocer los detalles internos de la implementación (a esto último también se denomina ***ocultación de información***)
  - Por ejemplo las estructuras de datos utilizadas internamente en una clase
- Si se modifican los detalles internos de la implementación no es necesario modificar ninguno de los clientes o métodos
- Los lenguajes Java y C++ utilizan las palabras reservadas: ***public***, ***protected*** y ***private*** para realizar el control de acceso o visibilidad de los miembros de una clase. C++ también soporta el concepto de clases y funciones amigas (***friends***)a las que se les permite el acceso a partes privadas.
- "*La ocultación es para prevenir accidentes, no para prevenir el fraude*".

[Stroustrup 1997, Capítulo 10, epígrafe 10.2.2]

Objetos (I)

Un objeto tiene estado, comportamiento e identidad; la estructura y comportamiento de objetos similares están definidos en su clase común; los términos "instancia" y "objeto" son intercambiables [Booch 94]



Un objeto tiene estado, exhibe algún comportamiento bien definido, tiene una identidad única.

Objetos (II)

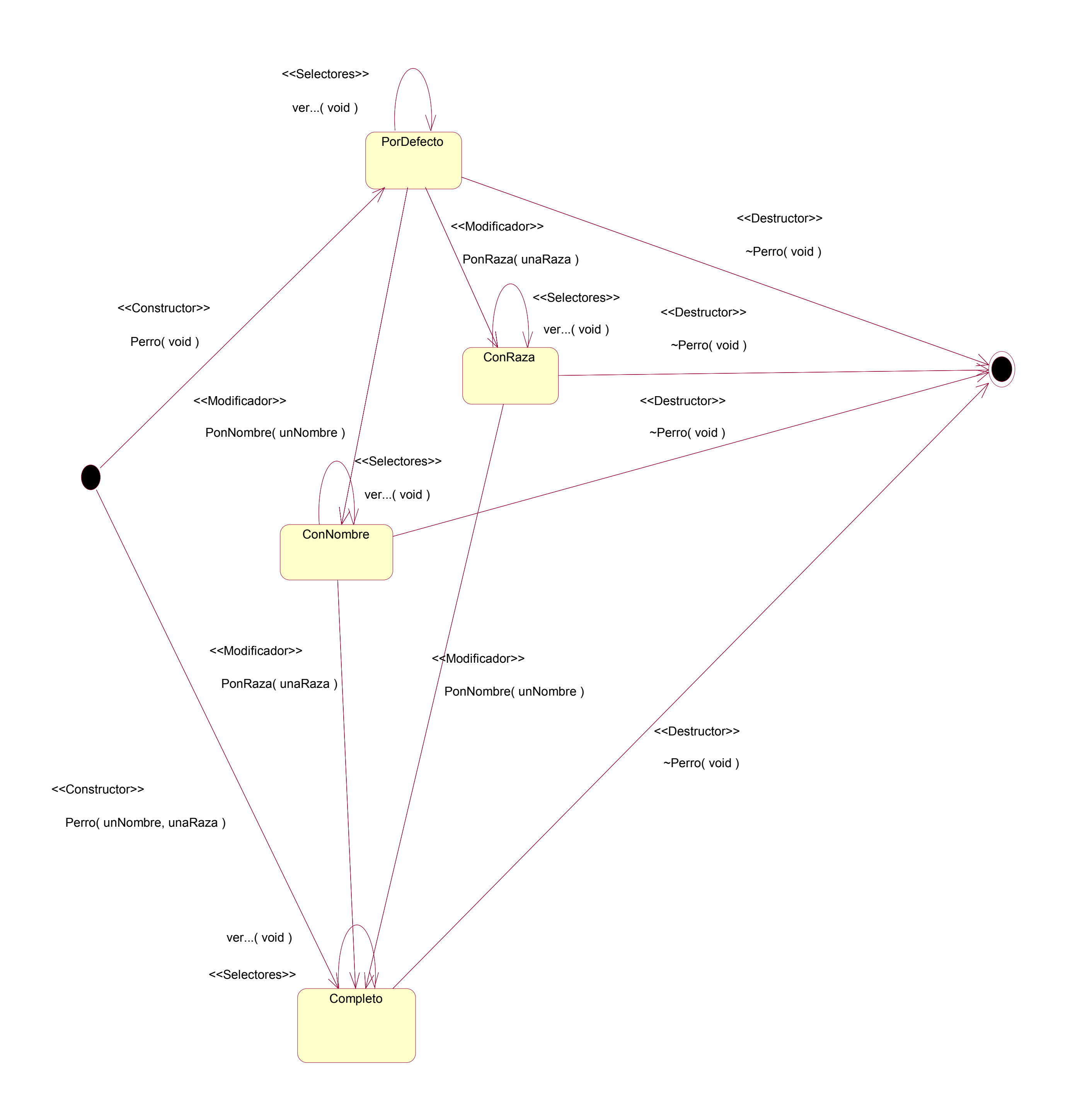
- **Estado**: *El estado de un objeto abarca todas las propiedades (normalmente estáticas) del mismo más los valores actuales (normalmente dinámicos) de cada una de esas propiedades*
- **Comportamiento:** *El comportamiento es cómo actúa y reacciona un objeto, en términos de sus cambios de estado y paso de mensajes.*
  – *El estado de un objeto representa los resultados acumulados de su comportamiento*
- **Operaciones:** *Una operación denota un servicio que una clase ofrece a sus clientes.*Los tipos de operaciones más frecuentes son:
  – **Modificador:** *Una operación que altera el estado de un objeto*
  – **Selector:** *Una operación que accede al estado de un objeto, pero no altera este estado*
  – **Iterador:** *Una operación que permite acceder a todas las partes de un objeto en algún orden perfectamente establecido*
  – **Constructor:** *Una operación que crea un objeto y/o inicializa su estado*
  – **Destructor:** *Una operación que libera el estado de un objeto y/o destruye el propio objeto*
- **Subprogramas libres:** En los lenguajes OO puros (Smalltalk, Eiffel, Java) sólo pueden declararse las operaciones como métodos. Sin embargo en los lenguajes OO híbridos (C++, Object Pascal, Ada, CLOS) pueden escribirse *subprogramas libres*, que en C++ se denominan *funciones no miembro,* y que en algunos casos se les puede dar ciertos privilegios al declararlas como amigas utilizando la palabra reservada *friend*.

Objetos (III)

- **Papeles (roles) y responsabilidades**
  - **protocolo:** *El protocolo es el caparazón externo que define el comportamiento admisible de un objeto.*
    - El protocolo engloba la visión estática y dinámica de un objeto
    - El protocolo está formado por los métodos de un objeto. En los lenguajes OO híbridos hay que añadir los subprogramas libres (en C++ las funciones amigas)
    - Para las abstracciones no triviales es útil dividir el protocolo en grupos lógicos de comportamiento denominados papeles (roles)
  - **papeles (roles):** *Los papeles son los grupos de comportamiento que desempeña un objeto y que constituyen un contrato entre una abstracción y sus clientes*
  - **responsabilidades:** *Las responsabilidades de un objeto son todos los servicios que proporciona para todos los contratos en los que está comprometido*
- **Objetos como autómatas finitos.** Algunos objetos se caracterizan por el orden en que se invocan sus operaciones, de tal forma que se pueden definir formalmente con un autómata finito.
- **Identidad:** *La identidad es aquella propiedad de un objeto que lo distingue de todos los demás objetos*
  - Los identificadores de los objetos pueden servir para distinguir objetos en un programa, pero no sirven para distinguir objetos persistentes cuya vida transciende a la del programa

Objetos (IV)

Diagrama de estados de los objetos de la clase *Perro*



Objetos (V)

- Los estados de los objetos se modelan con una combinación de valores de los atributos
- Los estados deben representar una situación en la vida del objeto
- No es buen diseño generalizar que cada estado se corresponda con el valor de un único atributo. Entre otras cosas por la gran cantidad de estados que se generarían

Tiempo de vida de un objeto

- *El **tiempo de vida** de un objeto se extiende desde el momento en que se crea por primera vez (y consume espacio por primera vez) hasta que ese espacio se recupera*
- *En **lenguajes con recolección de basura** (Java, Smalltalk, Eiffel, CLOS) un objeto se destruye automáticamente cuando todas las referencias a él se han perdido*
- *En **lenguajes con asignación y liberación explícita de memoria heap** (C++, Object Pascal, Ada) un objeto sigue existiendo y consume espacio incluso si todas las referencias a él han desaparecido*
- *Los **objetos persistentes** suelen implementarse utilizando una clase aditiva persistente de un marco de aplicación (framework). Así todos los objetos para los que se desea persistencia tienen a esta clase aditiva como superclase en algún punto de su trama de herencia de clases*

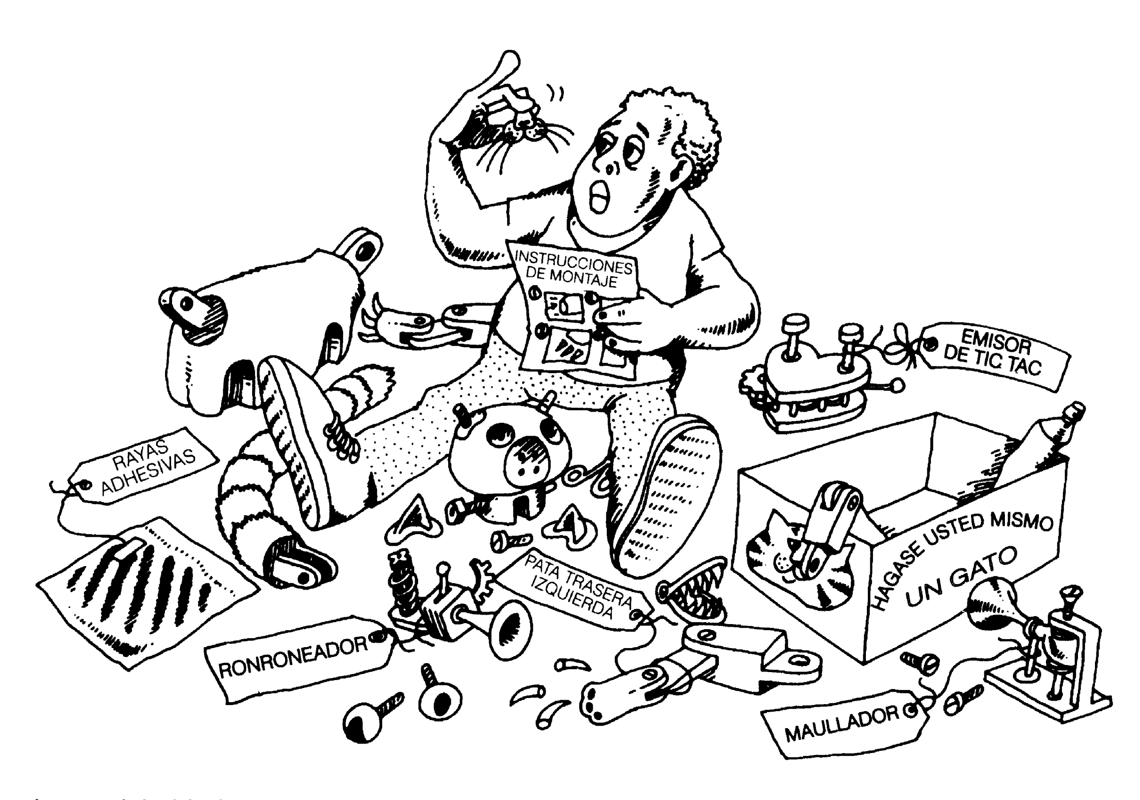
Tiempo de vida de un objeto

Diagramas de secuencia en UML



3.6 Modularidad

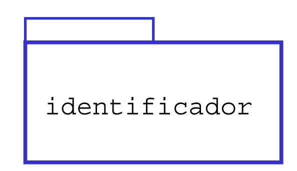
Es la propiedad que tiene un sistema que ha sido descompuesto en un conjunto de módulos cohesivos y débilmente acoplados *[Booch 94]*



La modularidad empaquetada las abstracciones en unidades discretas.

Modularidad (I)

- La modularidad es la propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes.
- Desde el punto de vista de la implementación la modularidad consiste en dividir una aplicación en módulos que se puedan compilar por separado.
- En UML el ***paquete*** es el mecanismo de propósito general para organizar elementos de modelado en grupos *[Booch 1999, capítulo 12*].



• En Java los módulos se denominan paquetes (*package*)

• En C++ los módulos son archivos compilados por separado. Habitualmente cada módulo consta de dos archivos:

– un archivo cabecera con la declaración de las clases (con extensión *.h* o *.hpp*)

–un archivo de implementación de los métodos de las clases (con extensión *.cpp*)

–Otro nivel de modularidad en C++ son los espacios de nombres (**namespaces**)

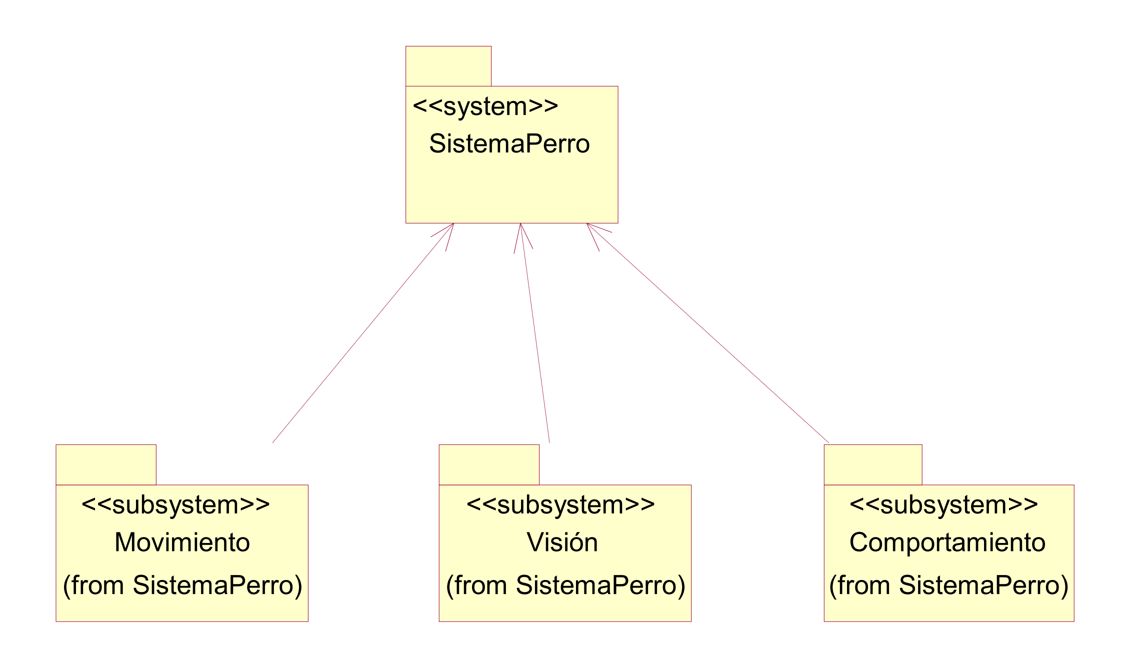
• En Ada cada módulo se define como un paquete (*package*) con dos partes: la especificación y el cuerpo del paquete

• En Object Pascal y Delphi los módulos se denominan *units* y son un archivo único.

• En C# se utilizan **namespaces**

Modularidad (II)

- Un paquete puede contener clases, interfaces, componentes,... Y otros paquetes
- UML define algunos estereotipos estándar que se aplican a los paquetes
  - `subsystem`
    - Especifica un paquete que representa una parte independientemente del sistema completo que se está modelando
  - `system`
    - Especifica un paquete que representa el sistema completo que se está modelando



Modularidad (III)

[Meyer 1997, capítulo 3]

El diseño modular se puede enunciar con 5 criterios, 5 reglas y 5 principios

- Los 5 **criterios**
  - Descomponibilidad
  - Componibilidad
  - Comprensibilidad
  - Continuidad
  - Protección
- Las 5 **reglas**
  - Correspondencia directa
  - Pocas interfaces
  - Interfaces pequeñas
  - Interfaces explícitas
  - Ocultación de la información
- Los 5 **principios**
  - Unidades modulares lingüísticas
  - Auto-documentación
  - Acceso uniforme
  - Principio abierto-cerrado
  - Elección única

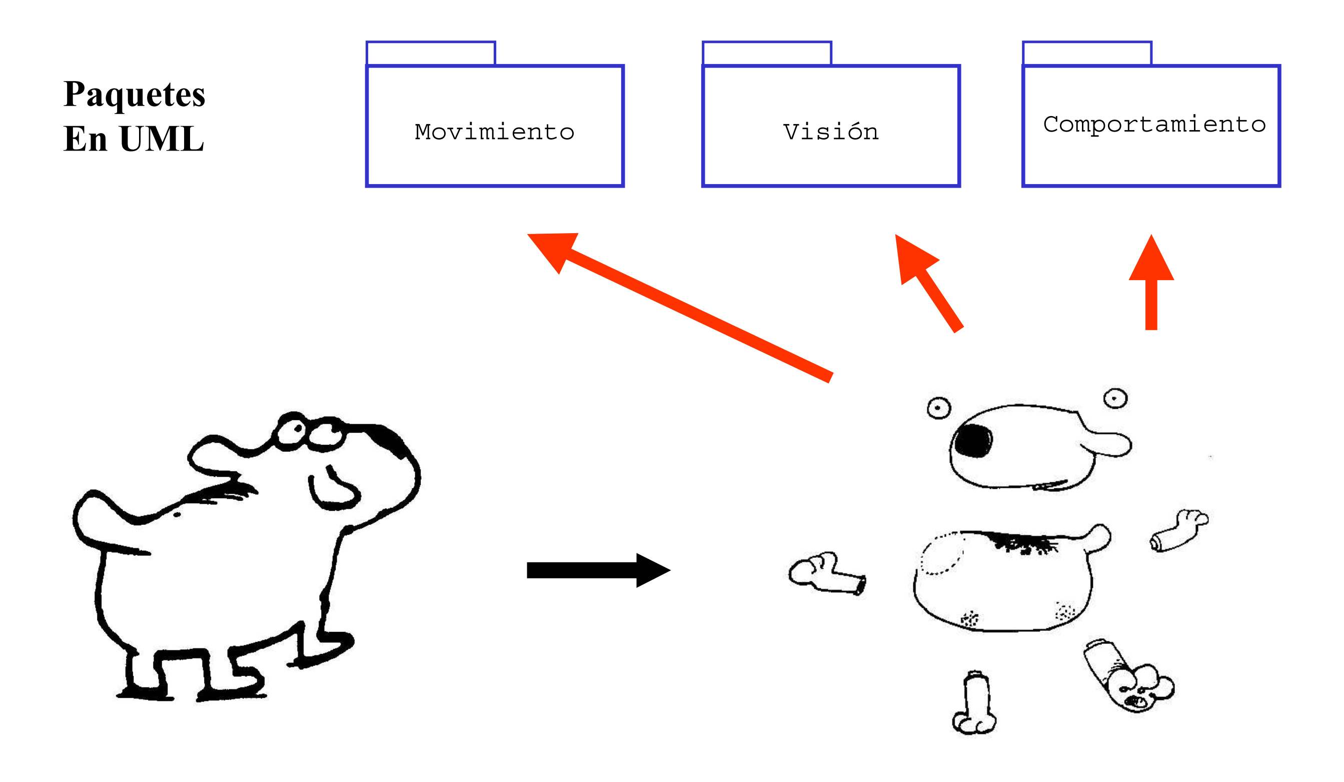
5 criterios de modularidad (I)

[Meyer 97, capítulo 3]

1º- Descomponibilidad modular

Es la tarea de separar un problema complejo en varios subproblemas menos complejos, conectados por una estructura simple y suficientemente independiente que permita construir cada subproblema separadamente

El ejemplo más típico de un método que satisface la descomponibilidad modular es el diseño descendente (*Top-down design*)

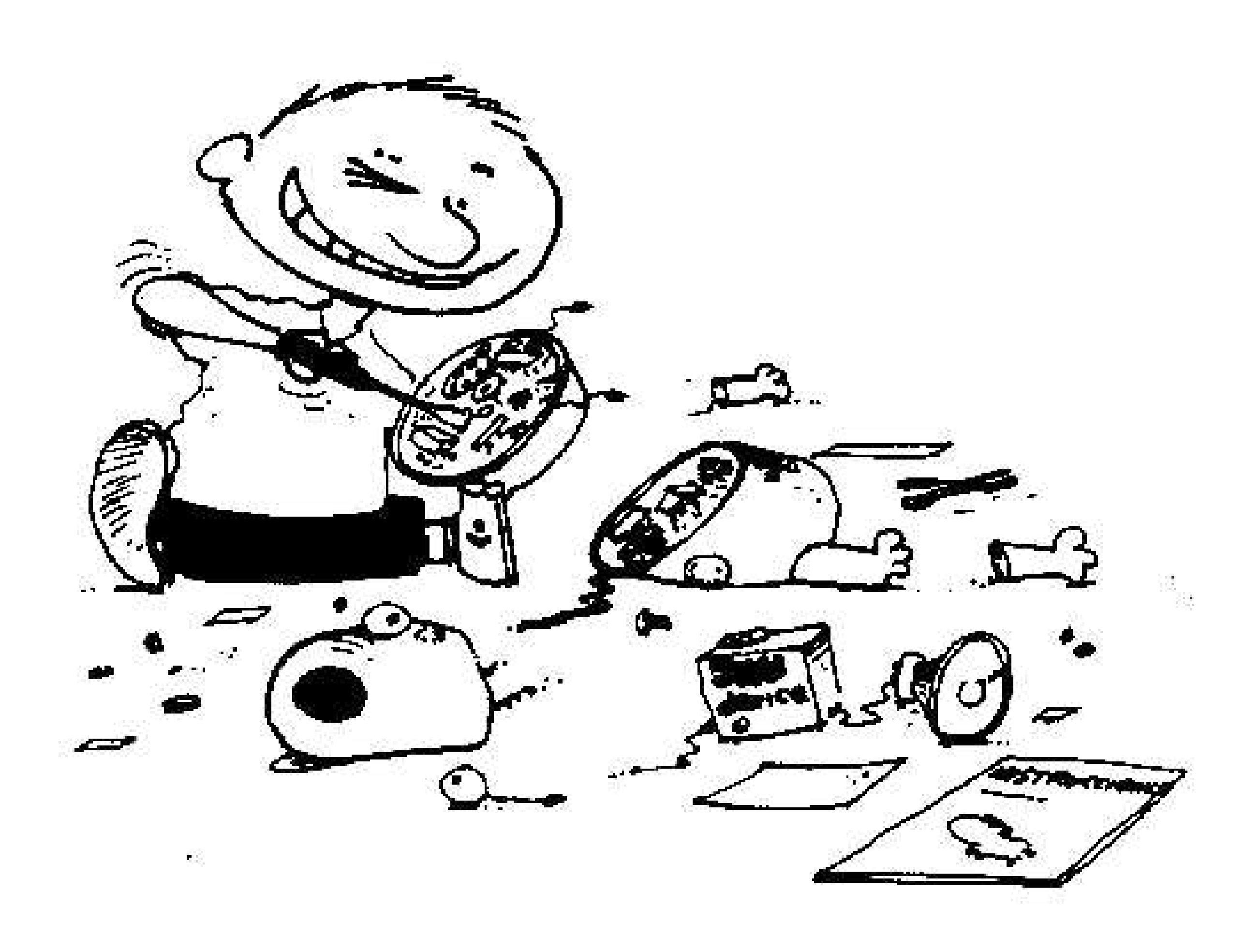


5 criterios de modularidad (II)

[Meyer 97 , capítulo 3]

2º - Componibilidad modular

Es la resolución de problemas por la composición de módulos previamente desarrollados



5 criterios de modularidad (III)

[Meyer 97 , capítulo 3]

3º - Comprensibilidad

Un método favorece la comprensibilidad modular si ayuda a producir software que permite comprender cada módulo por separado sin tener que conocer los otros, o en el caso peor tener que examinar sólo alguno de los otros módulos.

4º - Continuidad

Un método satisface la continuidad modular si un pequeño cambio en las especificaciones del problema obliga a modificar un módulo o un pequeño número de módulos.

5º - Protección

Un método satisface la protección modular si en el caso de que ocurra un anormal comportamiento en tiempo de ejecución éste se detecta en un módulo, o en el caso peor se propaga a unos pocos módulos vecinos.

5 reglas de modularidad (I)

[Meyer 97 , capítulo 3]

1º - Correspondencia directa

La estructura modular concebida en el proceso de construcción del sistema de software debe permanecer compatible con cualquier estructura modular creada en el proceso de modelado del dominio del problema

Esta regla sigue especialmente dos criterios:

- **Continuidad:** Conservando la traza de la estructura modular del problema será más fácil aquilatar y limitar los impactos de los cambios.
- **Descomponibilidad:** El trabajo realizado para analizar la estructura modular del dominio del problema puede ser un buen punto de comienzo para la descomposición modular del software.

5 reglas de modularidad (II)

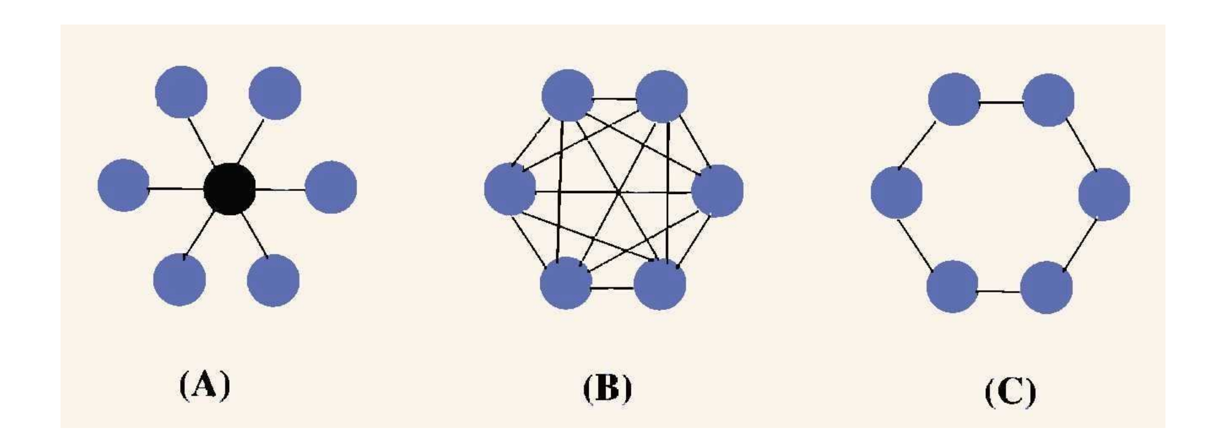
[Meyer 97 , capítulo 3]

2º - Pocas interfaces

Todo módulo debe comunicarse con el menor número de módulos posibles.

Esta regla sigue especialmente dos criterios:

- **Continuidad:** El efecto de los cambios se propaga al número menor de módulos
- **Protección:** El efecto de los errores se propaga al número menor de módulos



Si un sistema está compuesto de *n* módulos el número de conexiones entre módulos es:

• n-1	(fig. A) Mínimos enlaces, pero demasiado centralizada
• n(n-1)/2	(fig. B) Caso peor
• n	(fig. C) Tiene enlaces con los módulos vecinos, pero no está centralizado

5 reglas de modularidad (III)

[Meyer 97 , capítulo 3]

3º - Interfaces pequeñas

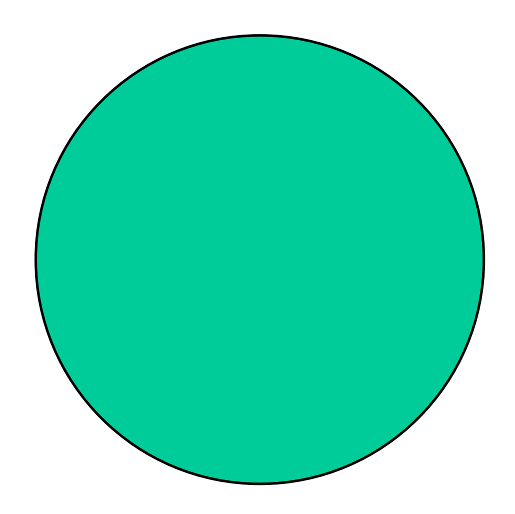
Si dos módulos intercambian información, esta debe ser la menor posible

Esta regla sigue especialmente dos criterios:

- **Continuidad:** El efecto de los cambios se propaga al número menor de módulos
- **Protección:** El efecto de los errores se propaga al número menor de módulos

Jeroglífico

¿ Por qué siempre que se desea dibujar a mano alzada un módulo suele pintarse un círculo?



Solución: Máxima información con mínima interfaz.
El círculo es la figura plana con máxima superficie para un perímetro dado

5 reglas de modularidad (IV)

[Meyer 97 , capítulo 3]

4º - Interfaces explícitas

Si dos módulos intercambian información, esta debe ser obvia desde la perspectiva de cada módulo y desde una perspectiva global a ambos.

Esta regla sigue especialmente los criterios:

- **Descomponibilidad:** Si es necesario descomponer un módulo en varios submódulos cualquier conexión entre ellos debe ser claramente visible
- **Componibilidad:** Si es necesario componer un módulo con varios submódulos cualquier conexión entre ellos debe ser claramente visible
- **Continuidad:** Facilita encontrar los elementos que pueden ser afectados por cambios.
- **Compresibilidad:** Facilita la comprensión de cada módulo y de todos los módulos relacionados con dicho módulo.

5 reglas de modularidad (V)

[Meyer 97 , capítulo 3]

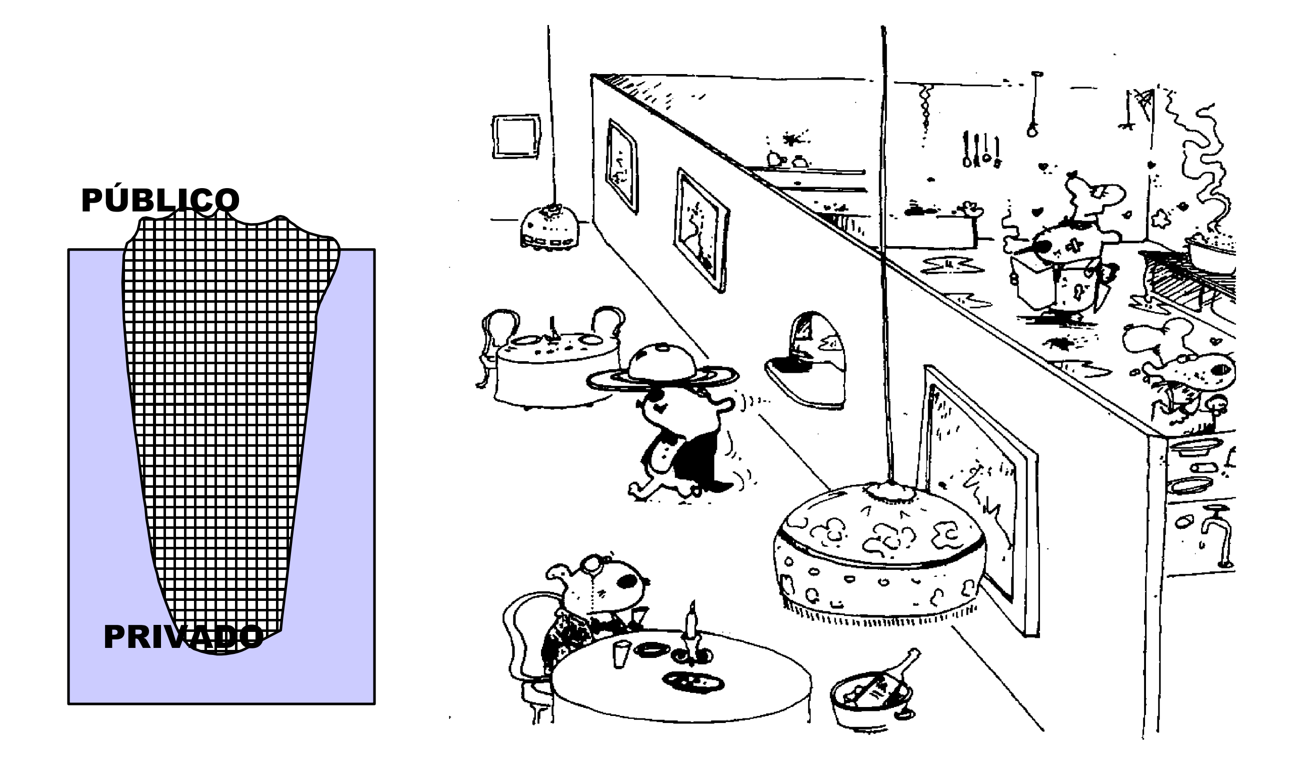
5º - Ocultación de información

El diseñador de un módulo debe seleccionar un subconjunto de propiedades de cada módulo como información oficial del módulo, esta información estará disponible para los autores de los módulos clientes.

El resto de la información es secreta.

Esta regla sigue especialmente el criterio:

- **Continuidad:** Los cambios de un módulo se tratarán de hacer en la parte secreta, pero no en la interfaz o parte pública.



5 principios de modularidad (I)

[Meyer 97 , capítulo 3]

1º - Unidades modulares lingüísticas

Los módulos deben corresponderse con unidades sintácticas en el lenguaje utilizado.

El lenguaje puede ser:

- *Un lenguaje de programación (los módulos deben poder compilarse por separado)*
- *Un lenguaje de diseño*
- *Un lenguaje de especificación,...*

Este principio sigue especialmente los criterios y reglas:

- **Continuidad:** Para asegurar la continuidad debe haber una correspondencia directa entre los módulos de especificación, diseño e implementación.
- **Correspondencia directa:** Se mantiene una correspondencia muy clara entre la estructura del modelo y la estructura de la solución.
- **Descomponibilidad:** Se asegura que cada módulo del sistema está bien delimitado en una unidad sintáctica.
- **Componibilidad:** La definición sintácticamente no ambigua facilita la combinación de módulos.
- **Protección:** Facilita el control del ámbito de los errores al estar los módulos delimitados sintácticamente.

5 principios de modularidad (II)

[Meyer 97 , capítulo 3]

2º - Auto-documentación

El diseñador de un módulo debe esforzase para hacer que toda la información relativa al módulo esté autocontenida en dicho módulo.

Este principio sigue especialmente los criterios:

- **Compresibilidad:** Es obvio que facilita la comprensión de cada módulo.
- **Continuidad:** Si el software y su documentación se tratan simultáneamente se garantiza que ambos permanezcan compatibles cuando las cosas comienzan a cambiar.

3º - Acceso uniforme

Todos los servicios ofertados por un módulo deben utilizarse a través de una notación uniforme, que no debe delatar como se implementa internamente ese servicio.

También se puede ver como un caso particular de la regla de ocultación de la información.

5 principios de modularidad (III)

[Meyer 97 , capítulo 3]

4º - Principio abierto-cerrado

Los módulos deben ser simultáneamente abiertos y cerrados.

Esta contradicción aparente responde a dos de diferente naturaleza:

- **Un módulo es abierto si se puede extender.** Por ejemplo utilizando la herencia.
- **Un módulo está cerrado si está disponible para ser utilizado por otros módulos.** Se supone que el módulo está definido con una descripción estable de su interfaz.

5º - Elección única

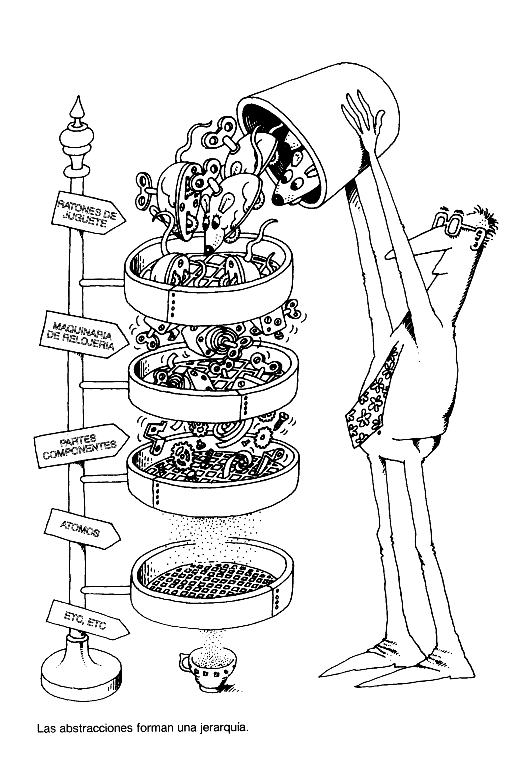
Siempre que un sistema de software deba soportar un conjunto de alternativas, uno y sólo un módulo debe conocer la lista exhaustiva de alternativas.

La tecnología de objetos ofrece dos técnicas conectadas con la herencia: el polimorfismo y el enlace dinámico.

3.7 Jerarquía

Es una clasificación u ordenación de abstracciones

[Booch 1994]



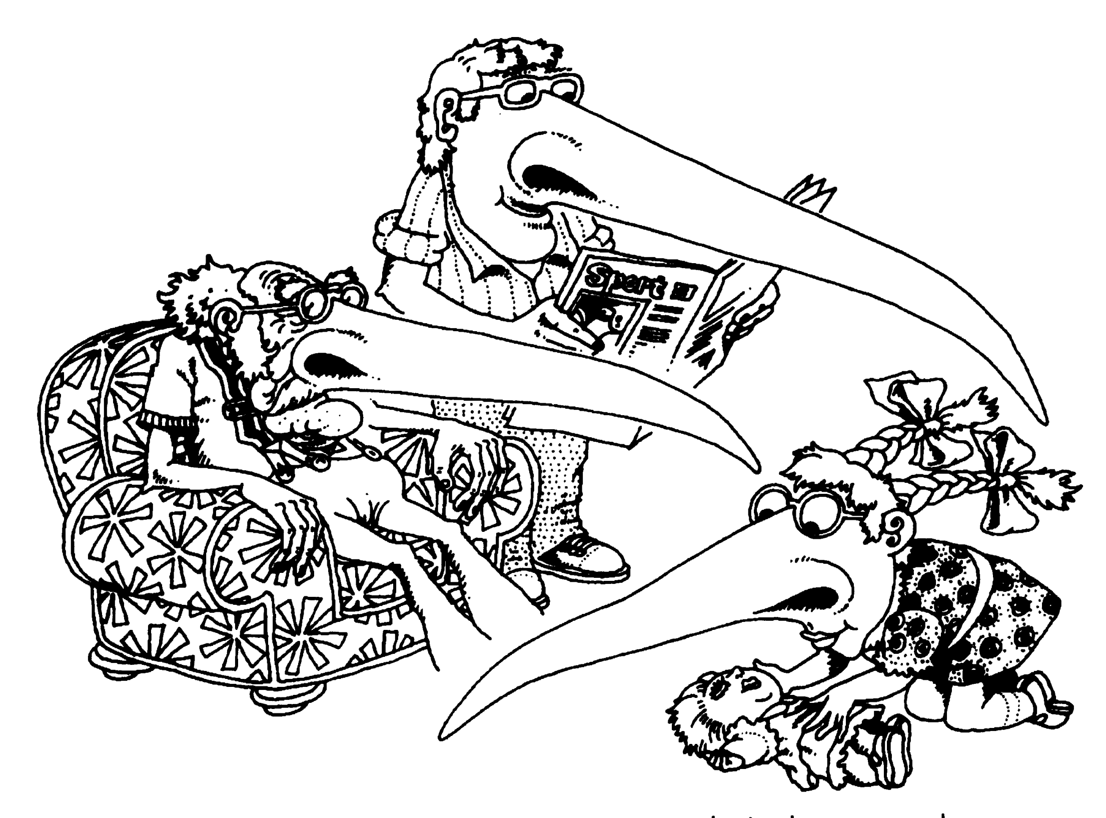
Las abstracciones forman una jerarquía.

Herencia

• *La herencia se debe utilizar para extender atributos y métodos dentro de una jerarquía.*

• *Sin embargo no debe verse como la única forma de trabajo, así la composición y la agregación permite delegar el trabajo a los objetos apropiados.*

• *También se denomina generalización (en UML), derivación (en C++ y C#) y extensión (en Java)*



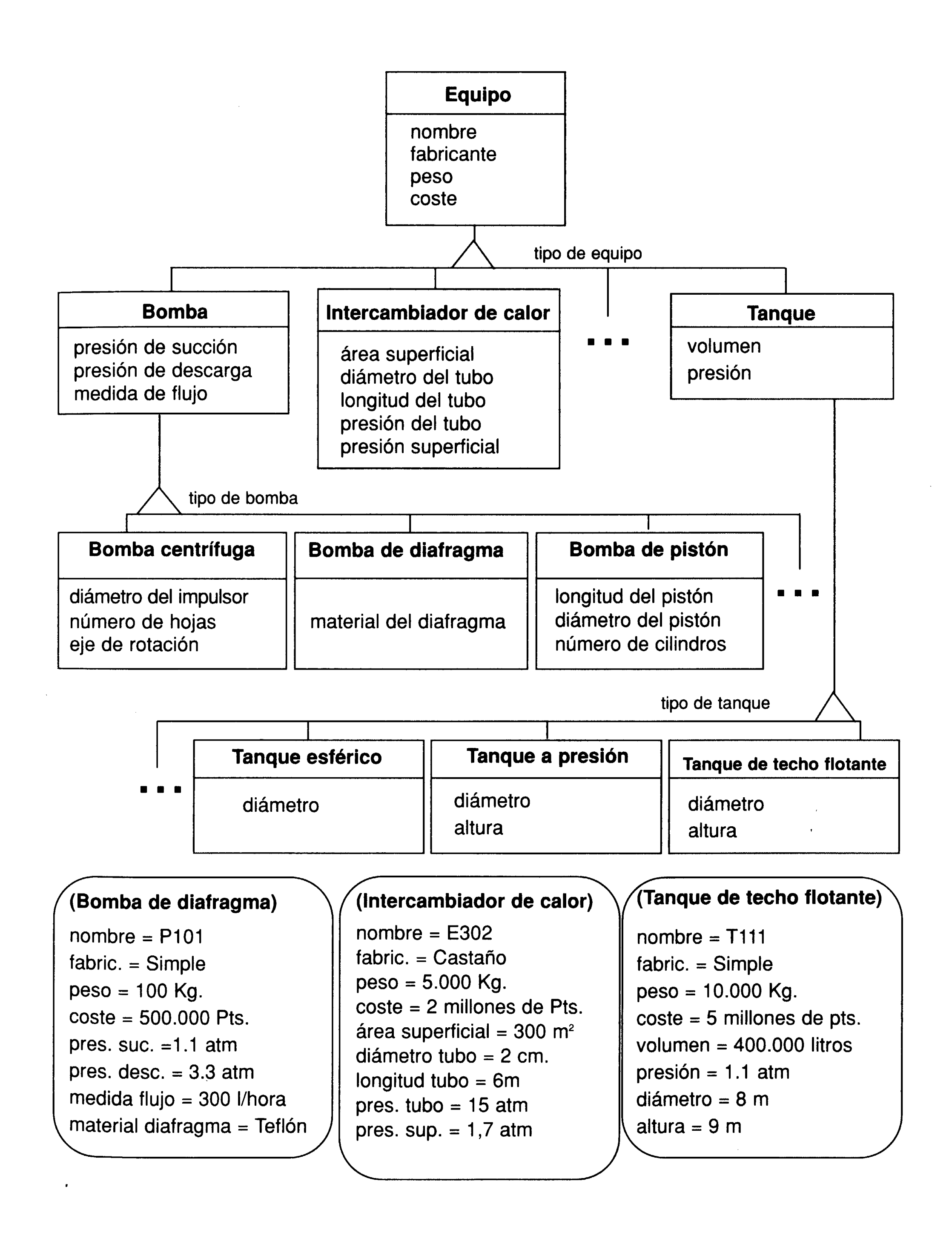
Una subclase puede heredar la estructura y comportamiento de su superclase.

[Booch 1994]

Herencia simple (I)

Ejemplo de herencia simple multinivel con instancias en notación *OMT*

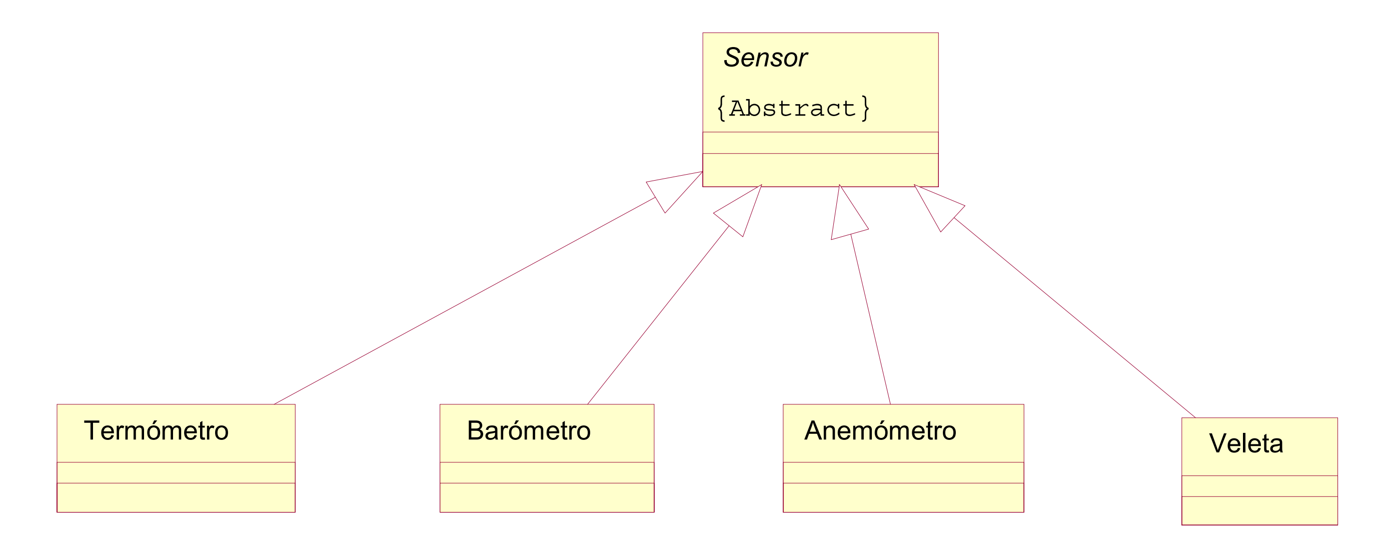
[Rumbaught 19 91]



Herencia simple (II)

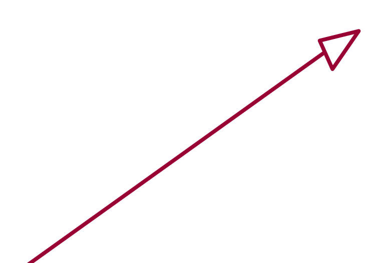
Ejemplo en UML

En UML a la herencia se la denomina generalización



• La **herencia** o **generalización** es una relación entre una clase general (llamada *superclase* o *padre*) y una clase más específica (llamada *hija* o *subclase*). La clase hija hereda todos los atributos y operaciones de la clase padre.

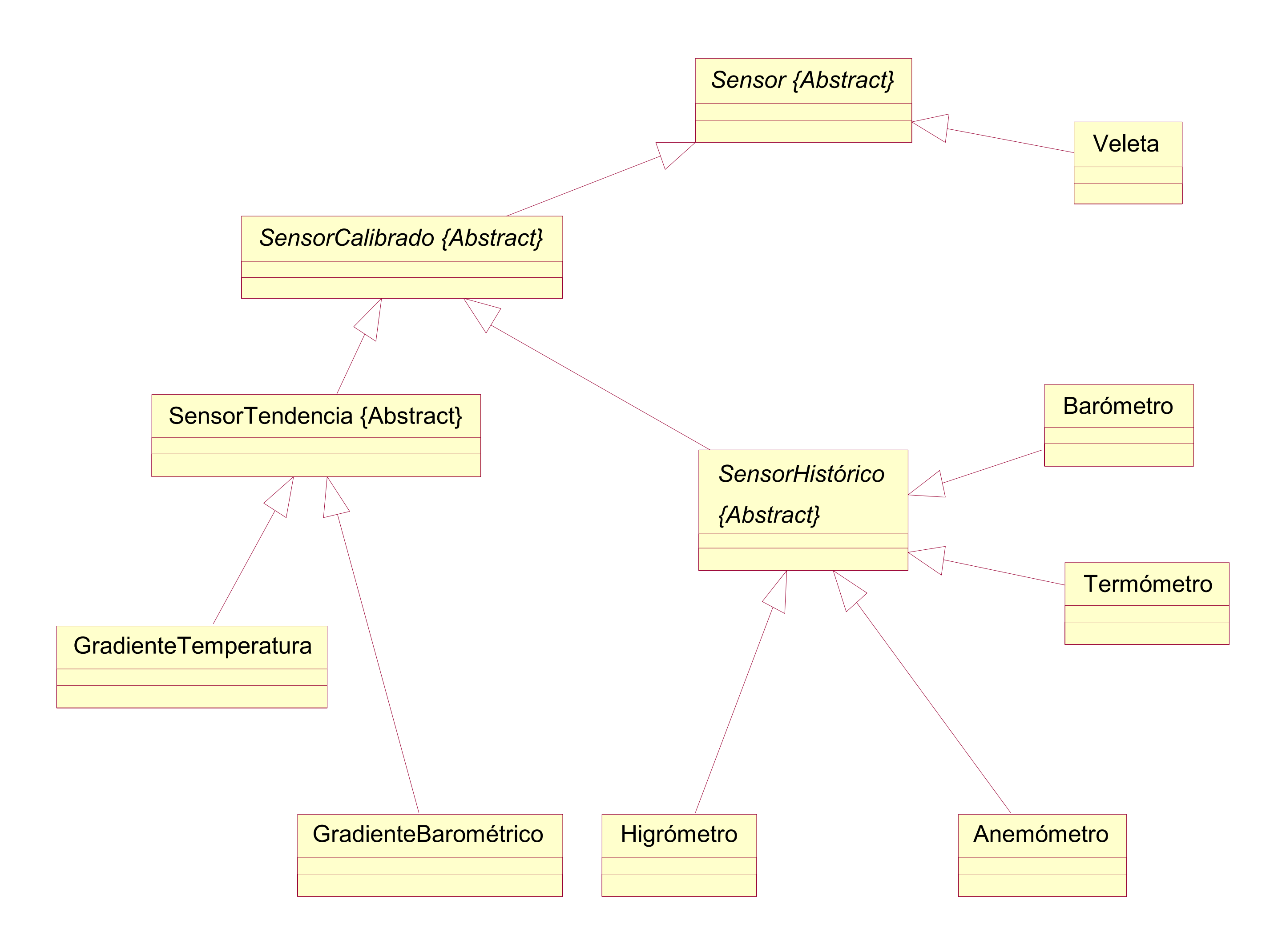
* Posteriormente se planteará el mismo ejemplo utilizando interfaces. Es decir Sensor también se puede plantear como una interfaz, que se implementará en las clases Termómetro, Barómetro,....



La herencia o generalización se representa con una flecha hueca en UML y se lee como "*Termómetro hereda de Sensor*" o como "*Sensor es una generalización de termómetro*"

Herencia simple y clases abstractas

Una clase abstracta es una clase que no tiene instancias, y que se diseña para ser clase base en una jerarquía de herencias



* Posteriormente se planteará el mismo ejemplo utilizando interfaces. Es decir Sensor, SensorCalibrado, ... también se puedne plantear como interfaces, que se implementarán en las clases Termómetro, Barómetro,....

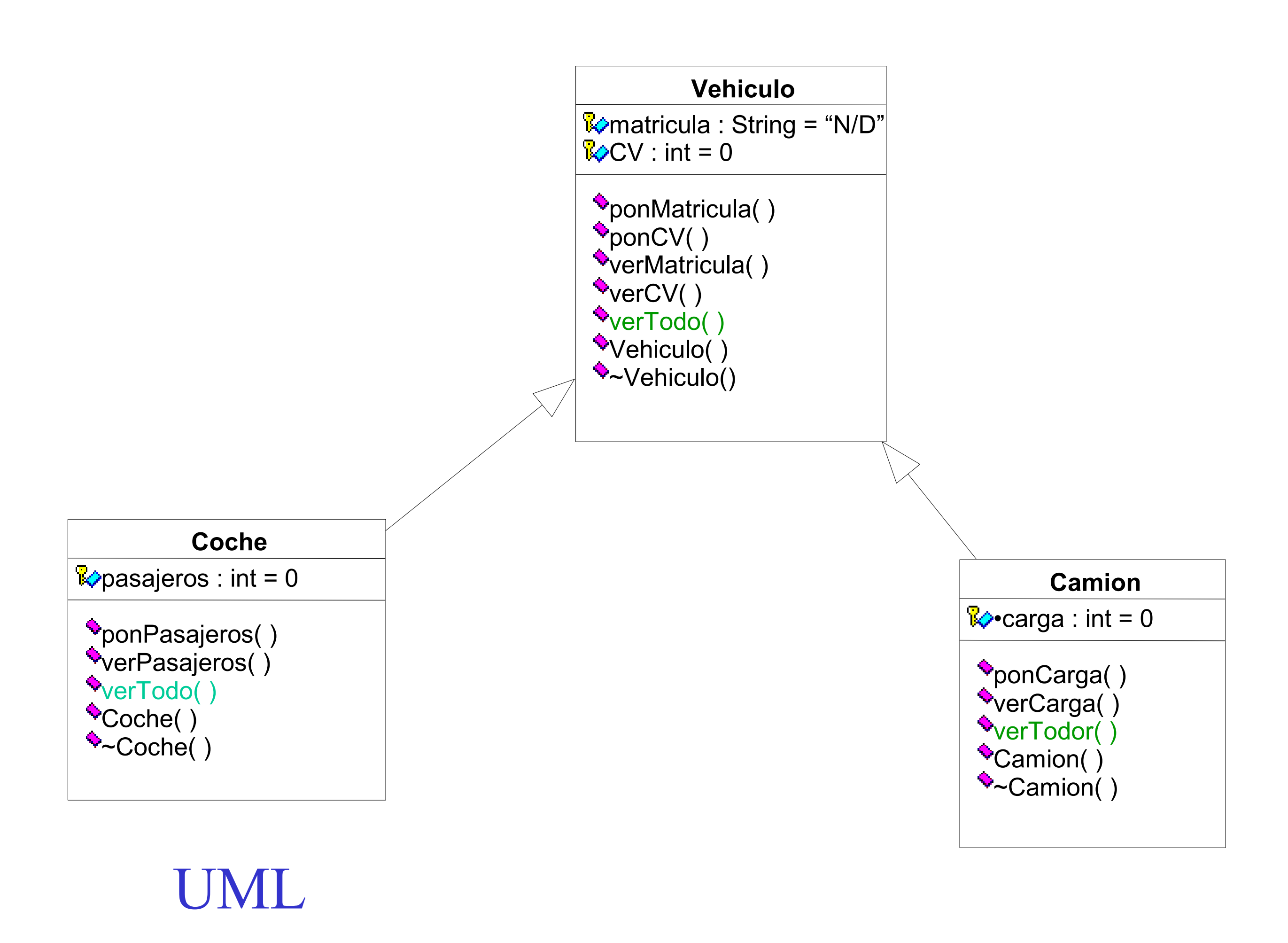
Herencia

- La herencia es una delas jerarquías de clases más importante y es uno de los elementos esenciales de los sistemas orientados a objetos. Aunque es necesaria manejarla con cuidado y sólo cuando es necesario.
- La herencia define una relación entre clases, en la que una clase comparte la estructura de comportamiento definida en una o más clases (*herencia simple o múltiple*)
- La herencia denota una relación "*es-un*" ("*is-a*"). Algunos autores también la denotan como "is-a-kind-of".
- La herencia implica una jerarquía de generalización/especialización, en la que una subclase especializa el comportamiento o estructura, más general, de sus superclases
- Las superclases representan abstracciones generalizadas
- Las subclases representan especializaciones en las que los atributos y métodos de la superclase. Pueden añadirse, modificarse o incluso ocultarse.
- La herencia puede violar los principios de la encapsulación
- Los lenguajes OO ofrecen mecanismos para acoplar encapsulación y herencia
- C++ ofrece que las interfaces de una clase sean partes
  - *private* que declaran miembros accesibles sólo a la propia clase
  - *protected* que declaran miembros accesibles a la propia clase y sus subclases
  - *public* que declaran miembros accesibles a todas las clases y funciones libres

Redefinición de métodos

(también denominada anulación o sustitución, en inglés *overriding)*

- Los atributos y métodos definidos en una superclase se heredan en las subclases.
- Los métodos de las subclases se pueden redefinir para que su comportamiento se adapte al de la subclase.
- En el siguiente ejemplo en UML *verTodo( )* está redefinido



UML

Implementación de la herencia en C++

[Stroustrup 2000, capítulo 12]

- El lenguaje C++ soporta herencia simple y múltiple.
- En C++ se habla de clases base y clases derivadas, Mientras que en otros lenguajes los modismos son clase padre y clases hijas, clase antepasado y clases descendientes, tipo y subtipo, etc.
- La sintaxis de la herencia en C++ es la siguiente:

```
class derivada : [acceso1] base1 [ , [acceso2] base2, ...]
    {
     // cuerpo
    }
```

- donde los corchetes indican opcionalidad y además
  - derivada: es el nombre de la clase que hereda de la clase base
  - base1, base2, ...: es la clase o clases de quién se hereda
  - acceso: es opcional y puede tomar los valores: **public** o **private.**
  - En ausencia de especificador de acceso se supone de tipo **private** cuando la clase está declarada como **class** y como de tipo **public** si está declarada como **struct**.
  - **public**. Cuando se hereda de una clase usando el especificador de acceso **public**, todos los elementos **public** de la clase base se convierten en elementos **public** en la clase derivada y los miembros **protected** de la clase base son miembros **protected** en la clase derivada
  - **private**. Cuando se hereda de una clase utilizando el especificador de acceso **private** los miembros **public** y **protected** son miembros de tipo **private** en la clase derivada
  - *[, [acceso2] base2, ...]* se utiliza para herencia múltiple

Ejemplo de herencia en C++ (I)

Definición de las clases de **Vehiculo.h (I)**

```
// Módulo Vehiculo.h (archivo cabecera)
// Clases: Vehiculo, Coche y Camion
// versión 2.0, 10 - Enero - 2002
// autor: J.M. Cueva Lovelle

#ifndef VEHICULO_HPP
#define VEHICULO_HPP

#include <iostream>
#include <string>     //Para manejo de string
using namespace std;  //Para usar la biblioteca estándar


//Clase base Vehiculo

class Vehiculo
{
protected:
  string matricula;
  int CV;
public:
  Vehiculo(void);                  // constructores sobrecargados
  Vehiculo(string unaMatricula, int unosCV);
  void ponMatricula(string unaMatricula) {matricula=unaMatricula;};
  string verMatricula(void) const {return matricula;};
  void ponCV(int unosCV){CV=unosCV;};
  int verCV(void)const {return CV;};
  void verTodo(void);
  ~Vehiculo(void)
    {cout<<"Se ha destruido el vehiculo:"<<verMatricula()<<endl;};
};

// La clase coche hereda de la clase vehiculo

class Coche: public Vehiculo
{
protected:
  int pasajeros;
public:
  Coche(void);                     // constructores sobrecargados
  Coche(string unaMatricula, int unosCV, int unosPasajeros);
  void ponPasajeros(int unosPasajeros){pasajeros=unosPasajeros;};
  int verPasajeros(void) const {return pasajeros;};
  void verTodo(void);
   ~Coche(void)
     {cout<<"Se ha destruido el coche:"<<verMatricula()<<endl;};
};
```

Ejemplo de herencia en C++ (II)

Definición de las clases de **Vehiculo.h**

```
// La clase camion hereda de la clase vehiculo

class Camion : public Vehiculo
{
 protected:
  int carga;
 public:
    Camion(); // constructores sobrecargados
    Camion (string unaMatricula, int unosCV, int unaCarga);
    void ponCarga(int unaCarga) {carga=unaCarga;};
    int verCarga(void)const {return carga;};
    void verTodo(void);
    ~Camion(void)
      {cout<<"Se ha destruido el camión:"<<verMatricula()<<endl;};
};


#endif  //fin VEHICULO_HPP
```

Ejemplo de herencia en C++ (III)

Implementación de las clases de **Vehiculo.cpp**

```
// Módulo Vehiculo.cpp
// Implementación de las clases: Vehiculo, Coche y Camion
// versión 2.0, 10 - Enero - 2002
// autor: J.M. Cueva Lovelle

#include "Vehiculo.h"

Vehiculo::Vehiculo(void)
{
            matricula="N/D";
            CV=0;
            cout<<"-----------------------------"<<endl;
            cout<<"Se ha construido un vehiculo"<<endl;
            cout<<"-----------------------------"<<endl;
}

Vehiculo::Vehiculo(string unaMatricula, int unosCV)
{
            matricula=unaMatricula;
            CV=unosCV;
            cout<<"-----------------------------"<<endl;
            cout<<"Se ha construido un vehiculo de matricula:";
            cout<<verMatricula()<<" con "<<verCV()<<" CV"<<endl;
            cout<<"-----------------------------"<<endl;
}

void Vehiculo::verTodo(void)
{
            cout<<"Vehiculo de matricula "<<verMatricula();
            cout<<" y "<<verCV()<<" CV"<<endl;
}
```

Ejemplo de herencia en C++ (IV)

Implementación de las clases de **Vehiculo.cpp**

```
// Implementación de la Clase Coche ------------------

Coche::Coche(void)
{
 //Llamada implícita al constructor por defecto de Vehiculo
 pasajeros=0;
 cout<<"------------------------------------"<<endl;
 cout<<"Se ha construido un coche"<<endl;
 cout<<"------------------------------------"<<endl;
}

Coche::Coche(string unaMatricula, int unosCV, int unosPasajeros)
{
 //Llamada implícita al constructor por defecto de Vehiculo
 matricula=unaMatricula;;
 CV=unosCV;
 pasajeros=unosPasajeros;
 cout<<"------------------------------------"<<endl;
 cout<<"Se ha construido un coche de matricula:";
 cout<<verMatricula()<<" y "<<verCV()<< " CV"<<endl;
 cout<<" con "<<verPasajeros()<< " pasajeros."<<endl;
 cout<<"------------------------------------"<<endl;
}

void Coche::verTodo(void)
{
Vehiculo::verTodo();
cout<<"con "<<verPasajeros()<<" pasajeros"<<endl;
}
```

Ejemplo de herencia en C++ (V)

Implementación de las clases de **Vehiculo.cpp**

```
// Implementación de la clase Camion --------------------

Camion::Camion(void)
{
 //Llamada implícita al constructor por defecto de Vehiculo
 carga=0;
 cout<<"------------------------------------"<<endl;
 cout<<"Se ha construido un camion"<<endl;
 cout<<"------------------------------------"<<endl;
}

Camion::Camion(string unaMatricula, int unosCV, int unaCarga)
{
 //Llamada implícita al constructor por defecto de Vehiculo
 matricula=unaMatricula;
 CV=unosCV;
 carga=unaCarga;
 cout<<"------------------------------------"<<endl;
 cout<<"Se ha construido un camion de matricula:";
 cout<<verMatricula()<<" y "<<verCV()<< " CV"<<endl;
 cout<<"con "<<verCarga()<< " toneladas de carga."<<endl;
 cout<<"------------------------------------"<<endl;
}

void Camion::verTodo(void)
{
 Vehiculo::verTodo();
 cout<<"con "<<verCarga()<<" toneladas de carga"<<endl;
}
```

Ejemplo de herencia en C++ (VI)

Prueba unitaria en GCC de las clases del módulo **Vehiculo.h**

```
// Prueba del módulo Vehiculo : PruebaVehiculo.cpp
// Ejemplo simple de herencia
// Versión 2.0, 10 - Enero - 2002
// Autor : J.M Cueva Lovelle
// Compilado con GCC (www.gnu.org) en linux
//$ g++ -o Pruebaehiculo.out PruebaVehiculo.cpp Vehiculo.cpp
// Para ejecutar $./PruebaVehiculo.out
#include "Vehiculo.h"
int main(int argc, char* argv[])
{
 Vehiculo moto;
 moto.verTodo();
 moto.ponMatricula("O-1234-AS");
 moto.ponCV(5);
 moto.verTodo();
 Vehiculo motoAgua("GI-4321-ZZ", 7);
 motoAgua.verTodo();

 Coche r11;
 r11.verTodo();
 r11.ponMatricula("4444CAC");
 r11.ponCV(8);
 r11.ponPasajeros(5);
 r11.verTodo();
 Coche r21("1234BKR", 12, 5);
 r21.verTodo();

 Camion pegaso;
 pegaso.verTodo();
 Camion roco("9876JMC",500,5);
 roco.verTodo();
 return 0;
}
```

Ejemplo de herencia en C++ (VII)

Prueba unitaria en C++Builder de las clases del módulo **Vehiculo.h**

```
// Prueba del módulo Vehiculo
// Ejemplo simple de herencia
// Versión 2.0, 10 - Enero - 2002
// Autor : J.M Cueva Lovelle
// Compilado con C++ Builder 4.0

#pragma hdrstop
#include <condefs.h>
#include "Vehiculo.h"
//---------------------------------------------------------------------------
// Para el uso de getch() se incluye conio
#include<conio>
//---------------------------------------------------------------------------
USEUNIT("Vehiculo.cpp");
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
 Vehiculo moto;
 moto.verTodo();
 moto.ponMatricula("O-1234-AS");
 moto.ponCV(5);
 moto.verTodo();

 Vehiculo motoAgua("GI-4321-ZZ", 7);
 motoAgua.verTodo();
```

Ejemplo de herencia en C++ (VIII)

Prueba unitaria en C++Builder de las clases del módulo **Vehiculo.h**

```
Coche r11;
 r11.verTodo();
 r11.ponMatricula("4444CAC");
 r11.ponCV(8);
 r11.ponPasajeros(5);
 r11.verTodo();

 Coche r21("1234BKR", 12, 5);
 r21.verTodo();

 Camion pegaso;
 pegaso.verTodo();

 Camion roco("9876JMC",500,5);
 roco.verTodo();

 cout<<"Pulse una tecla para finalizar programa" ;
 getch(); //Espera que se pulse una tecla
 return 0;
}
```

Ejemplo de herencia en C++ (IX)

Ejecución de la Prueba unitaria de las clases del módulo **Vehiculo.h**

```
-----------------------------
Se ha construido un vehiculo
-----------------------------
Vehiculo de matricula N/D y 0 CV
Vehiculo de matricula O-1234-AS y 5 CV
-----------------------------
Se ha construido un vehiculo de matricula:GI-4321-ZZ con 7 CV
-----------------------------
Vehiculo de matricula GI-4321-ZZ y 7 CV
-----------------------------
Se ha construido un vehiculo
-----------------------------
------------------------------------
Se ha construido un coche
------------------------------------
Vehiculo de matricula N/D y 0 CV
con 0 pasajeros
Vehiculo de matricula 4444CAC y 8 CV
con 5 pasajeros
-----------------------------
Se ha construido un vehiculo
-----------------------------
------------------------------------
Se ha construido un coche de matricula:1234BKR y 12 CV
 con 5 pasajeros.
------------------------------------
Vehiculo de matricula 1234BKR y 12 CV
con 5 pasajeros
-----------------------------
Se ha construido un vehiculo
-----------------------------
```

Ejemplo de herencia en C++ (X)

Ejecución de la Prueba unitaria de las clases del módulo **Vehiculo.h**

```
------------------------------------
Se ha construido un camion
------------------------------------
Vehiculo de matricula N/D y 0 CV
con 0 toneladas de carga
-----------------------------
Se ha construido un vehiculo
-----------------------------
------------------------------------
Se ha construido un camion de matricula:9876JMC y 500 CV
con 5 toneladas de carga.
------------------------------------
Vehiculo de matricula 9876JMC y 500 CV
con 5 toneladas de carga
Se ha destruido el camion:9876JMC
Se ha destruido el vehiculo:9876JMC
Se ha destruido el camion:N/D
Se ha destruido el vehiculo:N/D
Se ha destruido el coche:1234BKR
Se ha destruido el vehiculo:1234BKR
Se ha destruido el coche:4444CAC
Se ha destruido el vehiculo:4444CAC
Se ha destruido el vehiculo:GI-4321-ZZ
Se ha destruido el vehiculo:O-1234-AS
```

Implementación de la herencia en Java

(también denominada extensión de clases)

Vehiculo

VehiculoCarga

```
/**
 * clase VehiculoCarga, ilustra el manejo básico de la herencia
 * @version 1.0 15 de Julio 1998
 * @author Juan Manuel Cueva Lovelle
 */

class VehiculoCarga extends Vehiculo {
  protected float carga;

  //constructores
  VehiculoCarga(){
    super(); //invoca al constructor explícito de la clase padre
    }
  VehiculoCarga(String unaMatricula,
            String unPropietario,
            float unaVelocidadMaxima,
            int unNumeroPasajeros,
            float unaCarga){
            // Invoca al constructor de la clase padre
            super(unaMatricula, unPropietario,
                  unaVelocidadMaxima, unNumeroPasajeros);
            carga=unaCarga;
            }

  //métodos
  public float verCarga(){
      return carga;
      }

  // Redefinición o anulación de métodos

  public void verTodo(){
   super.verTodo(); //invoca al mismo método en la clase padre
   System.out.println( "Carga: " +verCarga());
   }
```

Implementación de la herencia en Java

Metodo *main* de la clase VehiculoCarga

```
public static void main(String[] args){

	// Se crean dos objetos

	VehiculoCarga cocheNuevo = new VehiculoCarga();
	VehiculoCarga renault = new VehiculoCarga("O-6333-AZ","JMCL",150,5,10);

	cocheNuevo.verTodo();
	renault.verTodo();


	// Prueba de métodos estáticos
	System.out.println("Número de vehículos producidos: "+
	                   VehiculoCarga.verVehiculosProducidos());

	// Prueba de toString
	System.out.println("Objeto cocheNuevo con toString: "+ cocheNuevo);
	System.out.println("Objeto renault con toString: "+ renault);

	// Creación de otro objeto
	VehiculoCarga otroCoche;

	// Aquí otroCoche es una referencia que contiene null

	otroCoche= new VehiculoCarga("O-9430-AS","JMCL",140,5,100);

	// con new se ha reservado memoria en el area heap

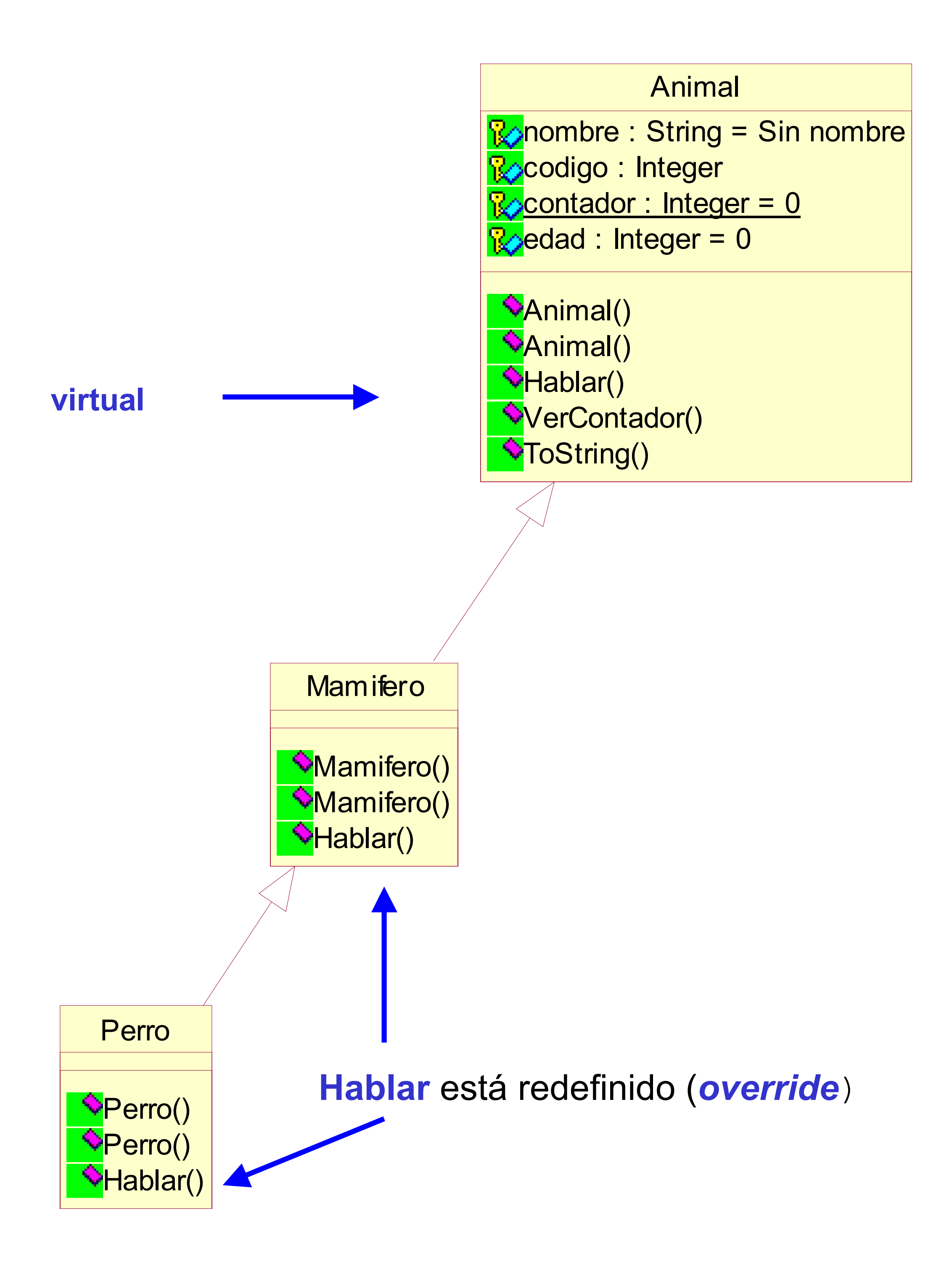
	otroCoche.verTodo();
	System.out.println( "Número de vehículos producidos: " +
	                   VehiculoCarga.verVehiculosProducidos());

	} // fin de main
}
```

Prueba unitaria de la clase *VehiculoCarga*

```
Matrícula: Sin matricular
Nombre del propietario: Sin propietario
Velocidad máxima: 0.0
Número de pasajeros: 0
Número de serie: 0
Carga: 0.0
Matrícula: O-6333-AZ
Nombre del propietario: JMCL
Velocidad máxima: 150.0
Número de pasajeros: 5
Número de serie: 1
Carga: 10.0
Número de vehículos producidos: 2
Objeto cocheNuevo con toString: Número de Serie -0(Sin matricular)
Objeto renault con toString: Número de Serie -1(O-6333-AZ)
Matrícula: O-9430-AS
Nombre del propietario: JMCL
Velocidad máxima: 140.0
Número de pasajeros: 5
Número de serie: 2
Carga: 100.0
Número de vehículos producidos: 3
```

Ejemplo de herencia en C# (I)



Ejemplo de herencia en C# (II)

```
// Animal.cs
// Versión 1.0
// Fecha 6-Enero-2003
// Autor Juan Manuel Cueva Lovelle
// Ejemplo de herencia

using System;

namespace Mascotas
{
    /// <summary>
    /// La clase animal es la base de una jerarquía de clases
    /// </summary>
    class Animal
    {
        protected string nombre="Sin nombre";
        /// <summary>
        /// Propiedad Nombre
        /// </summary>
        public string Nombre
        {
            set {nombre=value;}
            get {return nombre;}
        }
        protected int codigo;
        protected static int contador=0;

        // Propiedad Edad
        protected int edad;

        public int Edad
        {
            set
            {
                if (value>=0)
                    edad=value;
                else
                    Console.WriteLine("Edad no válida");
            }
            get{return edad;}
        }
        //Sobrecarga de constructores
        public Animal()
        {
            codigo=contador++;
            Console.WriteLine("Se construyo el animal "+nombre);
        }
```

Ejemplo de herencia en C# (III)

```
    public Animal(string nombre){
        this.nombre=nombre;
        codigo=contador++;
        Console.WriteLine("Se construyo el animal "+nombre);
    }
    public virtual void Hablar(){
        Console.WriteLine("El animal "+nombre+" dice um, um ...");
    }
    //Método estático
    public static int VerContador(){
        return contador;
    }
    public override string ToString(){
        return ("Animal:"+nombre+" Código:"+codigo+" Edad:"+edad);
    }
}
class Mamifero:Animal
{
    public Mamifero (){
        // Llamada implícita al constructor de la clase padre
        Console.WriteLine("Se construyo el mamifero "+nombre);
    }
    public Mamifero (string nombre):base(nombre){
        // Llamada implícita al constructor de la clase padre
        Console.WriteLine("Se construyo el mamifero "+nombre);
    }
    public override void Hablar(){
        base.Hablar();
        Console.WriteLine("El mamifero dice "+nombre+" dice uf,uf,...");
    }
}
class Perro:Mamifero{
    public Perro (){
        // Llamada implícita al constructor de la clase padre
        Console.WriteLine("Se construyo el perro "+nombre);
    }
    public Perro (string nombre):base(nombre){
        // Llamada implícita al constructor de la clase padre
        Console.WriteLine("Se construyo el perro "+nombre);
    }
    public override void Hablar(){
        base.Hablar();
        Console.WriteLine("El perro dice "+nombre+" dice guau");
    }
```

Ejemplo de herencia en C# (IV)

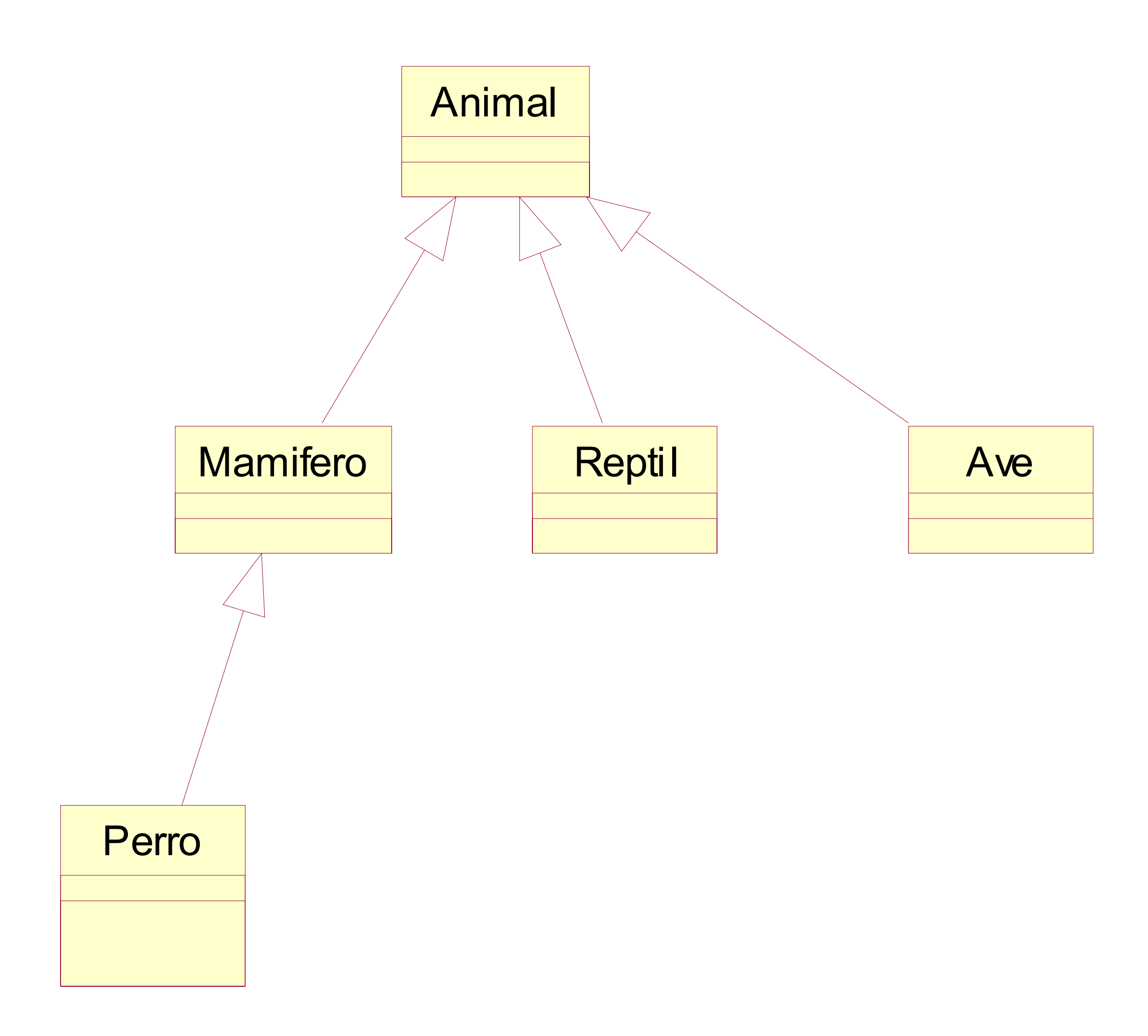
```
 /// <summary>
/// Prueba unitaria del módulo Mascotas
/// </summary>
[STAThread]
static void Main(string[] args)
{
    //Pruebas de la clase Animal
    Animal prehistorico = new Animal();
    Animal casero = new Animal("Elmo");
    prehistorico.Edad=5000;
    casero.Edad=1;
    casero.Hablar();
    string unNombre;
    Console.Write("Dame un nombre para un animal prehistórico: ");
    unNombre=Console.ReadLine();
    prehistorico.Nombre=unNombre;

    Console.WriteLine("Dime lo que sabes de ... "+prehistorico);
    Console.WriteLine("Dime lo que sabes de ... "+casero);

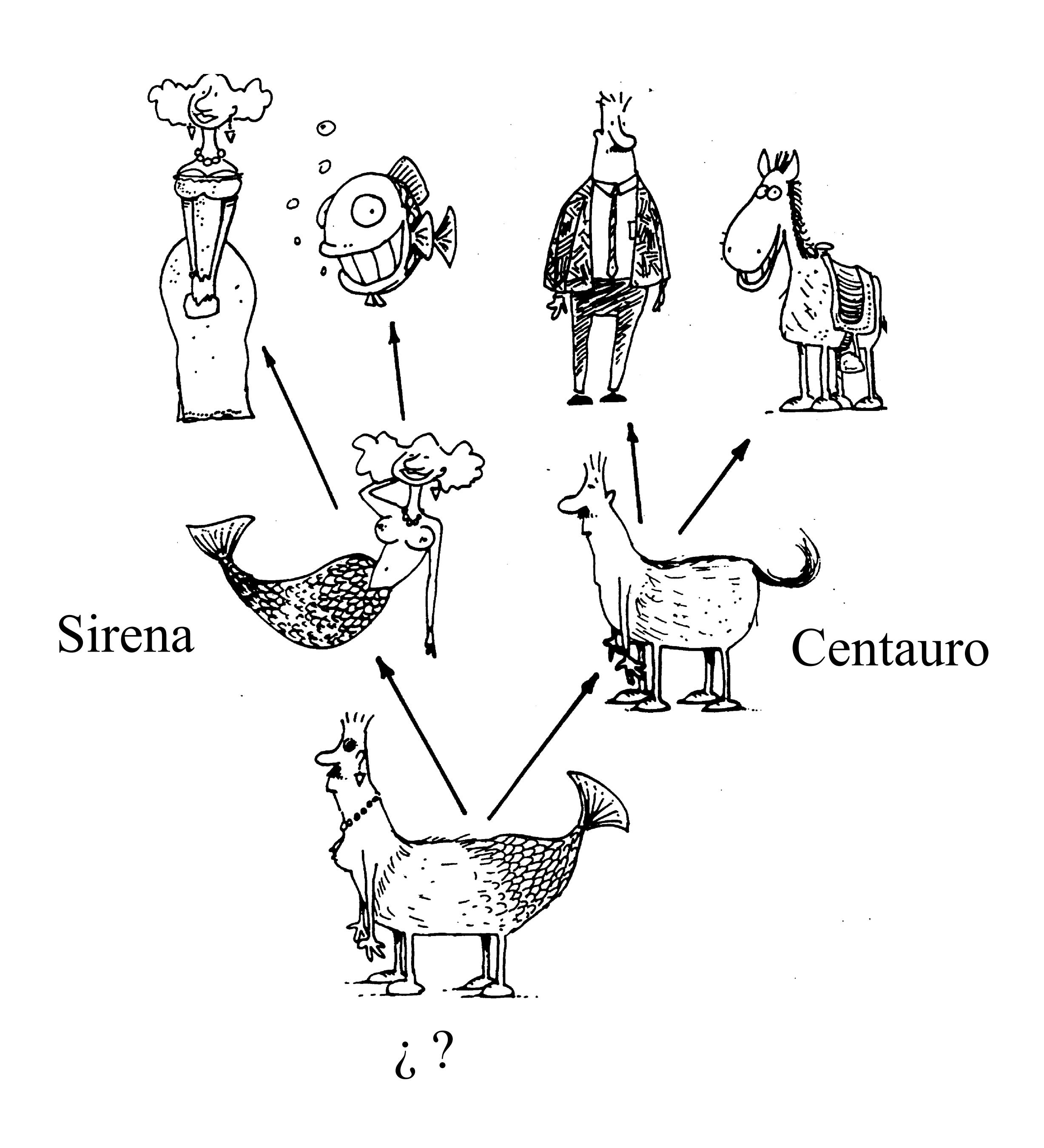
    // Pruebas de la clase Mamifero
    Mamifero delfin = new Mamifero();
    delfin.Nombre="delfín";
    Console.WriteLine("Dime lo que sabes de ... "+delfin);
    Mamifero ballena = new Mamifero ("baba");
    ballena.Hablar();
    Console.WriteLine("Dime lo que sabes de ... "+ballena);
    // Pruebas de la clase Perro
    Perro pio = new Perro();
    pio.Edad=2;
    pio.Nombre="pio";
    pio.Hablar();
    Console.WriteLine("Dime lo que sabes de ... "+pio);
    Perro pancho = new Perro ("pancho");
    pancho.Hablar();
    Console.WriteLine("Dime lo que sabes de ... "+pancho);
    //Ejemplo de llamada a método estático
    Console.WriteLine("Los animales creados son "+Animal.VerContador());
    //Espera una tecla para finalizar
    Console.ReadLine();
}
    }
}
```

Ejercicio de herencia simple

Realizar una implementación en C++, Java y C# de las siguientes clases. Hacer una prueba unitaria

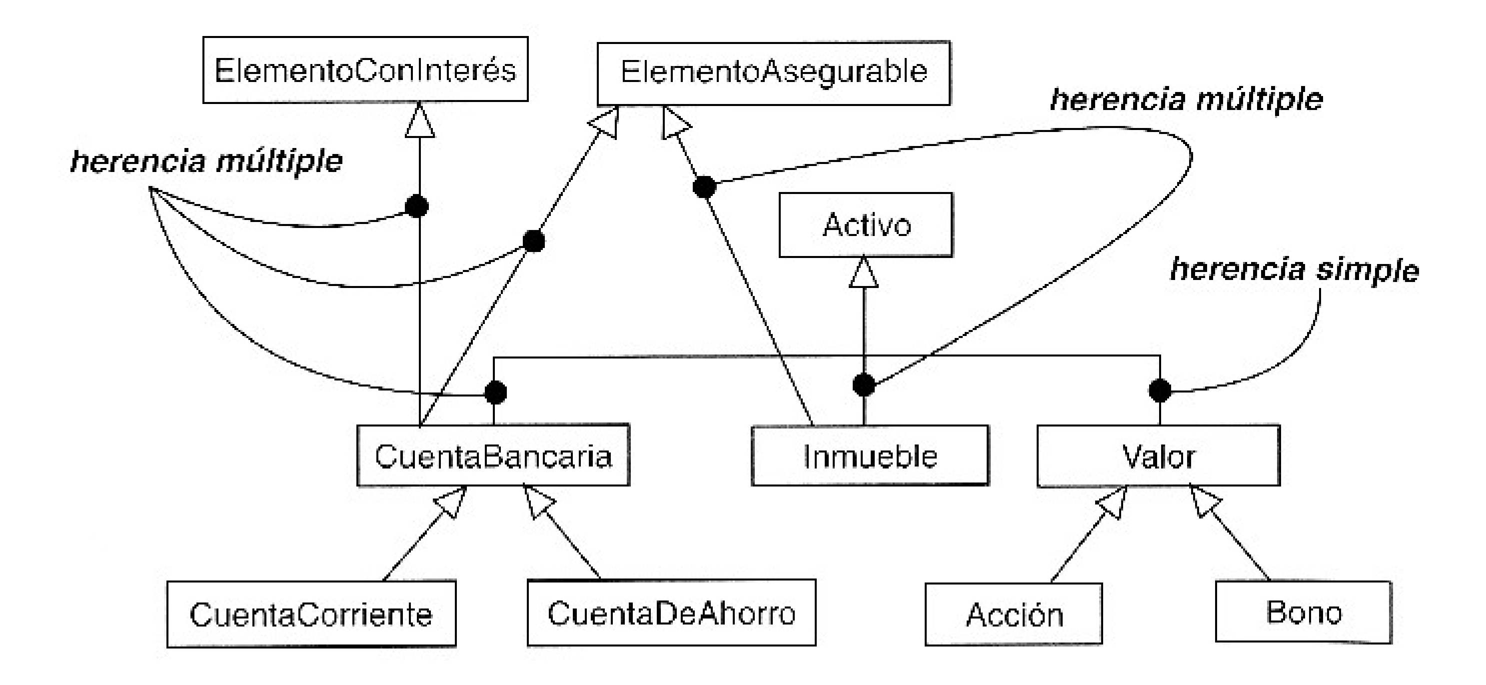


Herencia múltiple (I)

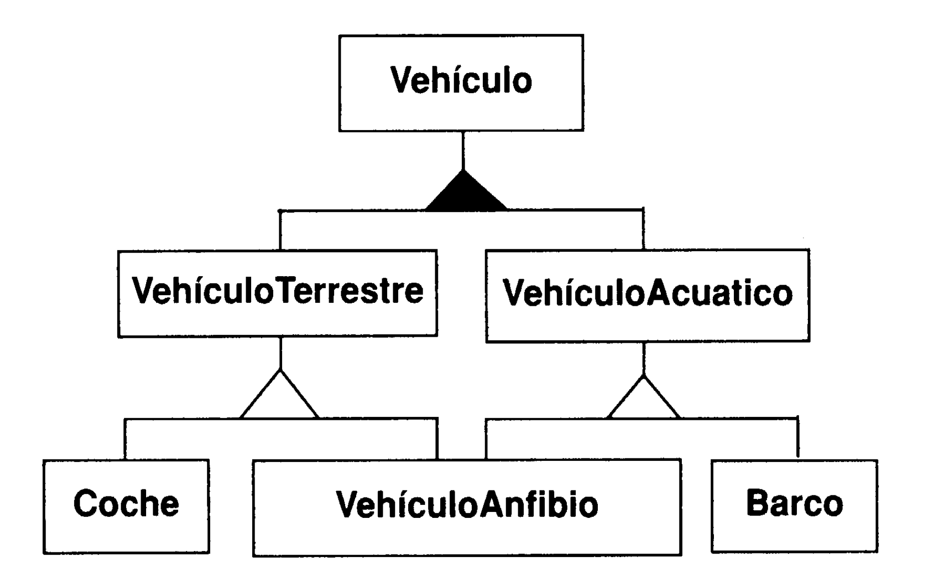


Herencia múltiple (II)

Ejemplo de herencia múltiple en UML *[Booch 1999]*



Herencia Múltiple (III)



- **Implementación en C++ (Versión 1)**

```
class Vehiculo {}
class VehiculoTerrestre: public Vehiculo {}
class VehiculoAcuatico: public Vehiculo {}
class Coche: public VehiculoTerrestre {}
class Barco: public VehiculoAcuatico {}
class VehiculoAnfibio: public VehiculoTerrestre, public VehiculoAcuatico {}
```

- **Implementación en C++ (Versión 2)**

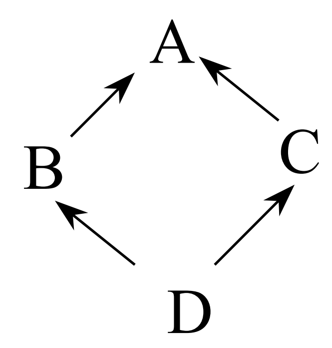
```
virtual class Vehiculo {}
class VehiculoTerrestre: public Vehiculo {}
class VehiculoAcuatico: public Vehiculo {}
class Coche: public VehiculoTerrestre {}
class Barco: public VehiculoAcuatico {}
class VehiculoAnfibio: public VehiculoTerrestre, public VehiculoAcuatico {}
```

- ¿ Herencia múltiple con repetición de atributos ?
  - Si la clase "*Vehiculo*" tiene el atributo "matricula"

    ¿Cuántos atributos "matricula" tiene un objeto de la clase "VehiculoAnfibio" ?

Herencia múltiple (IV)

- La herencia múltiple es conceptualmente correcta pero en la práctica introduce dos complejidades:
  - colisiones entre nombres de superclases diferentes
  - herencia repetida
- Las colisiones se producen cuando dos o más superclases proporcionan un campo u operación con el mismo nombre
  - En C++ se resuelven con calificación explícita
    - Ejemplo: `Sirena::Chica::cabeza,Sirena::Pez::cabeza`
  - En Smalltalk se utiliza la primera ocurrencia del nombre
- La herencia repetida ocurre cuando una clase D hereda de dos o más superclases “hermanas” (B y C) que a su vez heredan por algún camino de una clase (A) que es un antepasado común a ambas



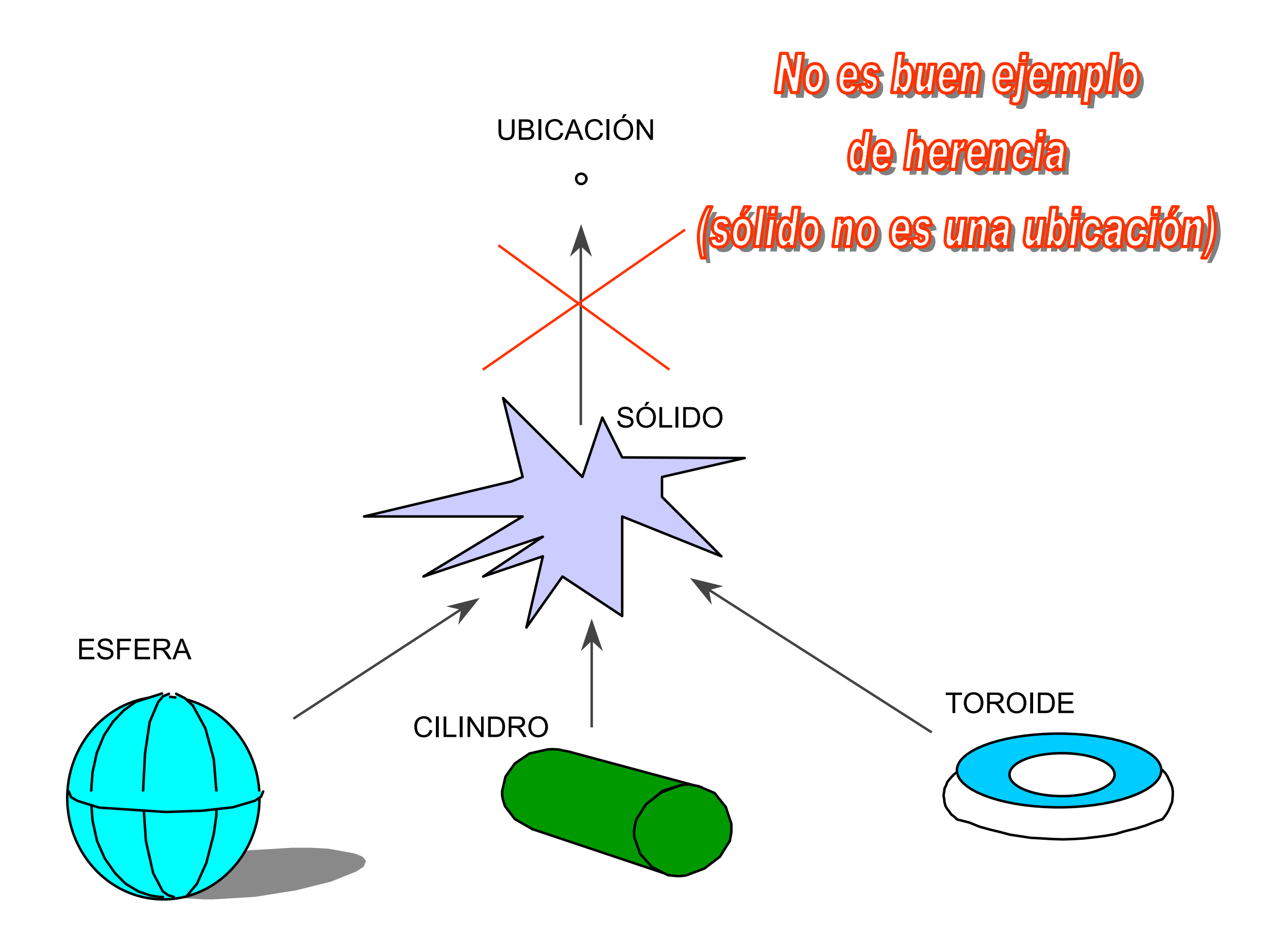
- ¿ Debe tener la clase que hereda de ambas una sola copia o muchas copias de la estructura de la clase compartida ?
- Algunos lenguajes prohiben la herencia repetida (Eiffel)
- C++ permite al programador que decida. Si las clase común es virtual no aparecen copias. Si no es virtual se producen copias repetidas, siendo necesaria la calificación explícita para resolver la ambigüedad al elegir las copias
  - Ejemplo: si `virtual class A {}` una sola copia.
- La herencia múltiple se sobreutiliza a menudo y es utilizada de forma inadecuada por principiantes
- Algunos lenguajes orientados a a enseñanza (por ejemplo Object Pascal) no permiten herencia múltiple obligando al principiante a utilizar siempre herencia simple. Java y C# sólo tiene herencia simple, sin embargo tiene implementación múltiple de interfaces

Riesgos de la herencia

[COAD 97, pp. 53]

- La herencia promueve una fuerte encapsulación con respecto a otras clases fuera de la jerarquía de herencia. Sin embargo la encapsulación es frágil entre las clases de la jerarquía, dado que un cambio en una superclase se trasmite a todas las subclases
- La herencia promueve un acomodo frágil a objetos que cambian de subclase con el tiempo

Jerarquía, abstracción y herencia



Interfaces

- Una interfaz es una colección de operaciones que se usa para especificar un servicio de una clase o de un componente
- Una interfaz no tiene atributos
- Gráficamente una interfaz se representa por un círculo
- De forma expandida se puede ver como una clase con el estereotipo `<<interface>>`
- Las operaciones sólo están especificadas, pero no implementadas
- Las operaciones pueden tener indicaciones sobre visibilidad y concurrencia
- Otra definición de Interfaces versus Clases es la dada por Bruce Eckel (www.bruceeckel.com)
  - Una interfaz es un TIPO y una clase es la IMPLEMENTACIÓN de un tipo.
  - Si la implementación es parcial nos encontramos con una clase abstracta
- Las interfaces pueden heredar de otras interfaces
- No existen interfaces en C++, pero si en Java y C#

Interfaces en UML



ISensor

Realización (forma sencilla)

ISensor

Termómetro

<<interface>>
Isensor

Activar()
Calibrar ()
Registrar()
Apagar()

estereotipo

Realización (forma expandida)

Veleta

operaciones

Polimorfismo

Es un mecanismo que permite a un método realizar distintas acciones al ser aplicado sobre distintos tipos de objetos que son instancias de una misma jerarquía de clases

- Polimorfismo significa "*muchas formas*"
- El polimorfismo se realiza en tiempo de ejecución gracias a la ligadura dinámica.
- No se debe confundir polimorfismo con sobrecarga.
- La sobrecarga se resuelve en tiempo de compilación, dado que los métodos se deben diferenciar en el tipo o en el número de parámetros.
- El polimorfismo se resuelve en tiempo de ejecución, todos los métodos tienen los mismos parámetros, las acciones cambian en función del objeto al que se le aplica.
- El lenguaje C++ utiliza los denominados *métodos virtuales* y punteros a objetos para implementar el polimorfismo.
- El lenguaje C# utiliza los denominados *métodos virtuales* y sus derivados redefinidos con *override* para implementar el polimorfismo.
- En lenguaje Java todos los métodos son virtuales por defecto y no hay la palabra reservada virtual.

Ejemplo de Polimorfismo en C++
Vehiculo.h

```
// Cabecera del Módulo Vehiculo (Vehiculo.h)
// Incluye las clases Vehiculo, Coche, Bicicleta
// Se utilizarán para ilustrar el polimorfismo
// Versión 1.0: 22-Noviembre-2000
// Autor: Juan Manuel Cueva Lovelle
//-----------------------------------------------
#ifndef VehiculoH
#define VehiculoH
#include <iostream>
using namespace std;
//-----------------------------------------------
class Vehiculo
{
 public:
 virtual void anda(void);
};

class Coche: public Vehiculo
{
 public:
 virtual void anda(void);
};

class Bicicleta: public Vehiculo
{
 public:
 virtual void anda(void);
};

#endif
```

Vehículo
----
anda ( )

Coche
----
anda ( )

Bicicleta
----
anda ( )

Ejemplo de Polimorfismo en C++
Vehiculo.cpp

```
// Implementación del Módulo Vehiculo (Vehiculo.cpp)
// Incluye las clases Vehiculo, Coche, Bicicleta
// Se utilizarán para ilustrar el polimorfismo
// Versión 1.0: 22-Noviembre-2000
// Autor: Juan Manuel Cueva Lovelle


#include "Vehiculo.h"


void Vehiculo::anda(void)
{
  cout<<"andando"<<endl;
};

void Coche::anda(void)
{
  cout<<"acelerando"<<endl;
};

void Bicicleta::anda(void)
{
  cout<<"pedaleando"<<endl;
};
```

Ejemplo de Polimorfismo en C++
Prueba.cpp

```
// Ejemplo de poliformismo
// Prueba del módulo Vehiculo
// Versión 1.0: 22-Noviembre-2000
// Compilado con g++ (GNU), de la siguiente forma:
// g++ -o Prueba.out Prueba.cpp Vehiculo.cpp
// Autor: Juan Manuel Cueva Lovelle

#include "Vehiculo.h"
int main()
{
 Vehiculo *v[5]; //array de 5 punteros a objetos

 v[0]= new Vehiculo;
 v[1]= new Coche;
 v[2]= new Bicicleta;
 v[3]= new Vehiculo;
 v[4]= new Coche;

 for (int i=0; i<5; i++)
   v[i]->anda();
delete v[];
return 0;
};
```

Ejecución

```
andando
acelerando
pedaleando
andando
acelerando
```

Explicación del código fuente

- Se crea un array polimórfico de punteros a objetos de la clase ***Vehiculo*** o cualquiera de sus descendientes.
- Dado que el array es de punteros, es necesario reservar memoria para crear los objetos, para eso se utiliza el operador **new**, que devuelve un puntero al objeto creado..
- Obsérvese que los punteros a objetos de una clase hija se pueden asignar a punteros a objetos de una clase padre (es una excepción de las reglas de comprobación de tipos)
- Se recorre el array polimórfico llamando al método virtual ***anda( ),*** que decide su comportamiento en tiempo de ejecución en función del objeto al que se le aplica.

Ejemplo de Polimorfismo en C++

Prueba.cpp (Versión C++Builder Consola)

```
// Ejemplo de poliformismo
// Prueba del módulo Vehiculo
// Versión 1.0: 22-Noviembre-2000
// Compilada con C++Builder 4.0
// Autor: Juan Manuel Cueva Lovelle

// Incluido por C++Builder
// hdrstop optimiza el uso de cabeceras precompiladas en C++Builder
// condefs.h se usa para aplicaciones de consola de C++Builder
// --------------------------
#pragma hdrstop
#include <condefs.h>
// --------------------------
// Para el uso de getch() se incluye conio.h
#include<conio>

#include "Vehiculo.h"
// Incluido por C++Builder para usar otro módulo
//-----------------------------------------------------------------
USEUNIT("Vehiculo.cpp");
//-----------------------------------------------------------------
// Incluido por C++Builder para que no dé el aviso (warning)
// de parámetros nunca usados
#pragma argsused
int main(int argc, char* argv[])
{
 Vehiculo *v[5];

 v[0]= new Vehiculo;
 v[1]= new Coche;
 v[2]= new Bicicleta;
 v[3]= new Vehiculo;
 v[4]= new Coche;

 for (int i=0; i<5; i++)
   v[i]->anda();
 cout<<"Pulse una tecla para finalizar programa" ;
 getch();
 delete v[];
 return 0;
};
```

Reglas del polimorfismo en C++

- Los objetos sobre los que se aplica el polimorfismo deben pertenecer a una misma jerarquía de clases
- Siempre tienen que usarse punteros o referencias a objetos.
- Se permite asignar un puntero (o referencia) a un objeto de una clase hija a un puntero (o referencia) de un objeto de una clase padre
- Los métodos virtuales deben ser idénticos en numero y tipo de parámetros entre la clase padre y sus descendientes.

Ejemplo de polimorfismo en Java (I)

```java
/**
   Ejemplo de polimorfismo
   @author Juan Manuel Cueva Lovelle
   @version 1.0 28-Septiembre-2000
*/

/**
  Ejemplo de clase Vehiculo2 para ilustrar el polimorfismo
*/

class Vehiculo2 {
           void anda(){
                    System.out.println("Andando");
           }
}

class Coche extends Vehiculo2 {
          void anda(){
                    System.out.println("Acelerando");
          }
}

class Bicicleta extends Vehiculo2 {
          void anda (){

          System.out.println("Pedaleando");
          }
}
```

Ejemplo de polimorfismo en Java (II)

```
public class PruebaPolimorfismo {
   public static void conduce (Vehiculo2 v){
                        v.anda();
// Polimorfismo:
// decide en tiempo de ejecución el anda() que ejecuta
} // Fin del método conduce

public static void main (String args[]){
int c;
System.out.println("Pulse C para crear un coche, B bicicleta, V vehiculo y X
salir");

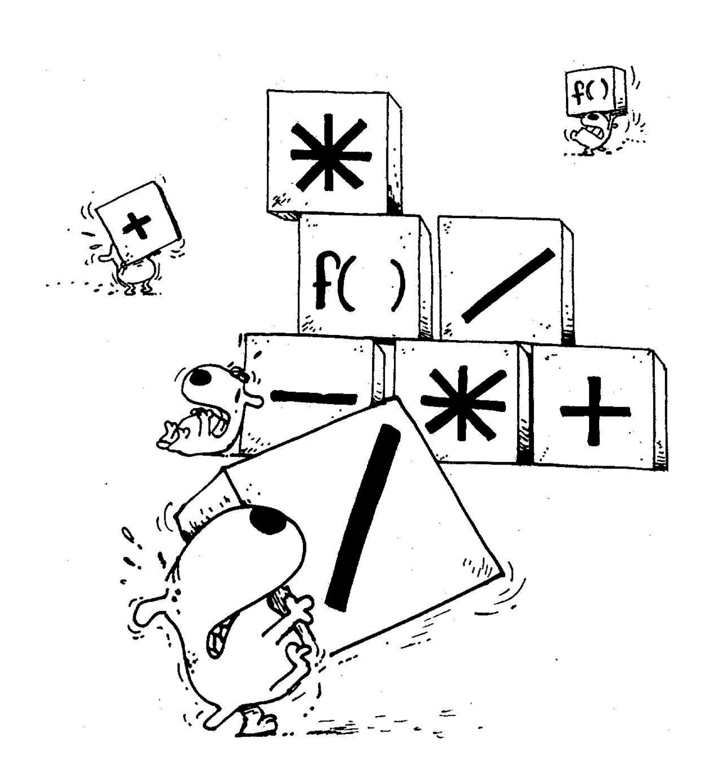
do {
 try {
   c=System.in.read();
   switch (c){
       case 'C': conduce(new Coche()); break;
       case 'B': conduce(new Bicicleta()); break;
       case 'V': conduce(new Vehiculo2()); break;
       case 10:
       case 13: break;
       default : System.out.println("Ha pulsado el caracter codificado como "+
c +" RECUERDE:Pulse C para crear un coche, B bicicleta, V vehiculo y X salir");
} // Fin del switch

  } catch (Exception e){
                             e.toString();
                             c='X';
 } // Fin del manejo de excepciones try-catch
} while(c!='X');
} // fin del método main
} // Fin de la clase Polimorfismo
```

Sobrecarga

(Overloading)

- Se produce cuando existen dos o más métodos con el mismo identificador pero con distinto número o tipo de parámetros resolviendo el compilador en tiempo de compilación.
- Algunos autores también denominan a la sobrecarga polimorfismo estático.
- Cada método realiza una operación en función de los argumentos que recibe
- Algunos lenguajes como C++ también soportan sobrecarga de operadores, en este caso el significado de cada operador depende del operando al que se aplique.



Sobrecarga

Restricciones a la sobrecarga de operadores en C++

- No se puede alterar la precedencia de ningún operador
- No se puede alterar el número de operandos que requiere cada operador. Aunque el operador puede no usar algún operando para nada.
- Los operadores sobrecargados pueden ser heredados por cualquier clase hija o derivada. Excepto el operado asignación (=)
- Los únicos operadores que no se pueden sobrecargar son :
  - Operador punto .
  - Operador resolución de ámbito ::
  - Operador indirección .*
  - Operador ternario ? :

Sobrecarga

Ejemplo

Vector
x : float
y : float
z : float
nombre : String
Vector( )
Vector( )
+( )
-( )
=( )
verNombre( )
ponX( )
ponY( )
ponZ( )
verVector( )
verVector( )
ponNombre( )
~Vector( )
ponXYZ( )

Sobrecarga. Ejemplo en C++.vector.h (I)

```
//Clase Vector (vector.h)
//Ejemplo de sobrecarga de operadores
// Versión 2.0: 14-Enero-2002. Autor: Juan Manuel Cueva Lovelle
#ifndef VECTOR_H
#define VECTOR_H
#include <iostream>
#include <string>     //Para manejo de string
using namespace std;  //Para usar la biblioteca estándar
class Vector {
protected:
  float x, y, z; // vector tridimensional
  string nombre;
public:
  Vector(void); // Constructores sobrecargados
  Vector(float, float, float, string);
  Vector operator+(Vector); // Sobrecarga de operadores
  Vector operator-(Vector);
  Vector operator=(Vector);
  void verVector(int); //Sobrecarga de Métodos
  void verVector(void){cout << verNombre();verVector(1);};
  string verNombre(void){return nombre;};
  void ponX(float unaX){x = unaX;}; // Modificadores
  void ponY(float unaY){y = unaY;};
  void ponZ(float unaZ){z = unaZ;};
  void ponXYZ(float unaX, float unaY, float unaZ)
        {x = unaX; y = unaY; z = unaZ;}
  void ponNombre(string unNombre){nombre=unNombre;};
  ~Vector(void);
} ;
#endif
```

Sobrecarga. Ejemplo en C++ . vector.cpp(II)

```
// Implementación de la Clase Vector (vector.cpp)
// Ejemplo de sobrecarga de operadores
// Versión 2.0: 14-Enero-2002
// Autor: Juan Manuel Cueva Lovelle
#include "vector.h"
// Sobrecarga de constructores
Vector::Vector(void)
{
 x=0;y=0;z=0;
 nombre="Nulo";
 cout <<"Se ha construido el vector "<<verNombre()<<endl;
}
Vector::Vector(float mx, float my, float mz, string unNombre)
{
  x = mx;
  y = my;
  z = mz;
  nombre=unNombre;
  cout <<"Se ha construido el vector "<<verNombre()<<endl;
}
// Sobrecarga del operador +
Vector Vector::operator+(Vector t)
{
  Vector temp;
  temp.x = x + t.x;
  temp.y = y + t.y;
  temp.z = z + t.z;
  return temp;
}
```

Sobrecarga. Ejemplo en C++.vector.cpp (III)

```
// Sobrecarga del operador -
Vector Vector::operator-(Vector t)
{
  Vector temp;
  temp.x = x - t.x;
  temp.y = y - t.y;
  temp.z = z - t.z;
  return temp;
}
// Sobrecarga de =
Vector Vector::operator=(Vector t)
{
  x = t.x;
  y = t.y;
  z = t.z;
  return *this;
}
void Vector::verVector(int i)
{
 // el parámtro i sólo se usa para la sobrecarga
  cout << "("<< x << ", ";
  cout << y << ", ";
  cout << z << ")"<<endl;
}
Vector::~Vector(void)
{
 cout <<"Se ha destruido el vector "<<verNombre()<<endl;
}
```

Sobrecarga. Ejemplo en C++. Sobrecarga.cpp (IV)
Prueba unitaria de la clase Vector
Compilado con GCC (www.gnu.org)

```cpp
// Ejemplo de Sobrecarga (Sobrecarga.cpp)
// Versión 2.0 : 14-Enero-2002
// Autor: Juan Manuel Cueva Lovelle
// Compilado con GCC en linux
// $g++ -o Sobrecarga.out Sobrecarga.cpp vector.cpp
#include "vector.h "

int main(void)
{
  Vector a, b(1,2,3,"B"), c(4,5,6,"C");
  a.verVector(); b.verVector(); c.verVector();
  c = a+b;
  c.verVector();
  c = a+b+c;
  c.verVector();
  b.ponX(100); b.ponY(200); b.ponZ(300);
  c=b-c;
  c.verVector();
  a.ponXYZ(10,20,30);
  a.ponNombre("A");
  c = b = a;  // asignación múltiple
  a.verVector();
  c.verVector();
  b.verVector();
return 0;
};
```

Sobrecarga. Ejemplo en C++. Sobrecarga.cpp (V)
Prueba unitaria de la clase Vector. Compilado con C++ Builder

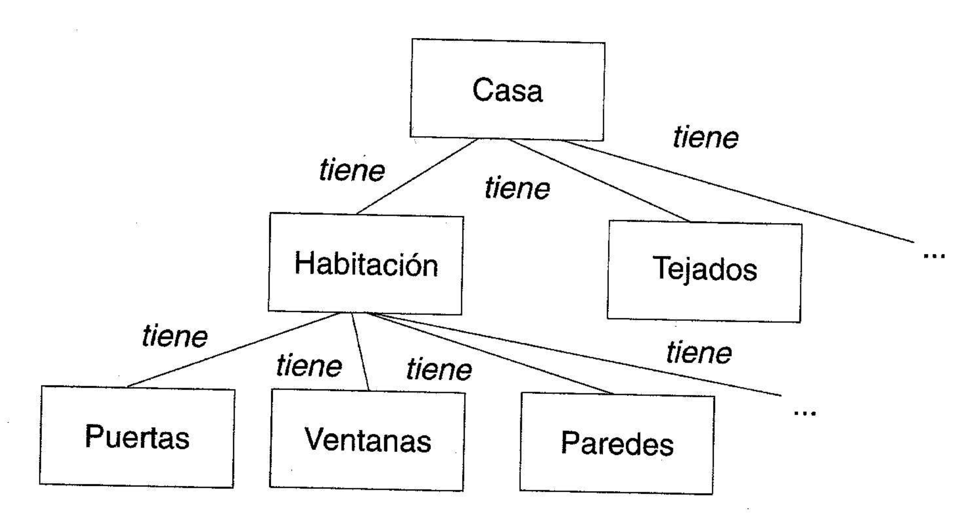
```
// Ejemplo de Sobrecarga (Sobrecarga.cpp)
// Versión 2.0: 14-Enero-2002
// Compilada con C++Builder 4.0
// Autor: Juan Manuel Cueva Lovelle
#pragma hdrstop
#include <condefs.h>
// Para el uso de getch() se incluye conio.h
#include<conio.h>
#include "vector.h"
USEUNIT("vector.cpp");
#pragma argsused
int main(int argc, char* argv[])
{
  Vector a, b(1,2,3,"B"), c(4,5,6,"C");
  a.verVector(); b.verVector(); c.verVector();
  c = a+b;
  c.verVector();
  c = a+b+c;
  c.verVector();
  b.ponX(100); b.ponY(200); b.ponZ(300);
  c=b-c;
  c.verVector();
  a.ponXYZ(10,20,30);
  a.ponNombre("A");
  c = b = a;  // asignación múltiple
  a.verVector();
  c.verVector();
  b.verVector();
 cout<<"Pulse una tecla para finalizar programa" ;
 getch();
 return 0;
};
```

Sobrecarga. Ejemplo en C++ (VI)
Ejecución de la prueba unitaria de la clase Vector

```
Se ha construido el vector Nulo
Se ha construido el vector B
Se ha construido el vector C
Nulo(0, 0, 0)
B(1, 2, 3)
C(4, 5, 6)
Se ha construido el vector Nulo
Se ha destruido el vector Nulo
Se ha destruido el vector B
Se ha destruido el vector Nulo
Se ha destruido el vector C
C(1, 2, 3)
Se ha construido el vector Nulo
Se ha destruido el vector Nulo
Se ha destruido el vector B
Se ha construido el vector Nulo
Se ha destruido el vector Nulo
Se ha destruido el vector C
Se ha destruido el vector Nulo
Se ha destruido el vector C
Se ha destruido el vector Nulo
C(2, 4, 6)
Se ha construido el vector Nulo
Se ha destruido el vector Nulo
Se ha destruido el vector C
Se ha destruido el vector Nulo
Se ha destruido el vector C
C(98, 196, 294)
Se ha destruido el vector A
Se ha destruido el vector B
Se ha destruido el vector C
A(10, 20, 30)
C(10, 20, 30)
B(10, 20, 30)
Se ha destruido el vector C
Se ha destruido el vector B
Se ha destruido el vector A
```

Agregación y composición

- Las relaciones de agregación y composición son jerarquías “*parte de*” o “*tiene un*”
- La composición y agregación no es un concepto exclusivo de los lenguajes orientados a objetos. Por ejemplo el uso de estructuras anidadas en C.
- La agregación y la composición plantean el problema de la propiedad, vida y relaciones entre los componentes agregados y la clase contenedora que los contiene.
- La agregación es un concepto simple con una semántica profunda
- La **agregación** no liga las vidas de la clase contenedora y de las partes que la componen.
  - Se representa una relación con rombo hueco
- La **composición** es una forma de agregación con una fuerte relación de pertenencia y vidas coincidentes de las partes con el todo.
  - Las partes con una multiplicidad no fijada pueden crearse después de la parte compuesta a la que pertenecen, pero una vez creadas viven y mueren con ella.
  - Se representa una relación con un rombo relleno



Agregación y composición en UML
Asociación, cardinalidad y roles

Agregación

Composición

Asociación

0..1 *

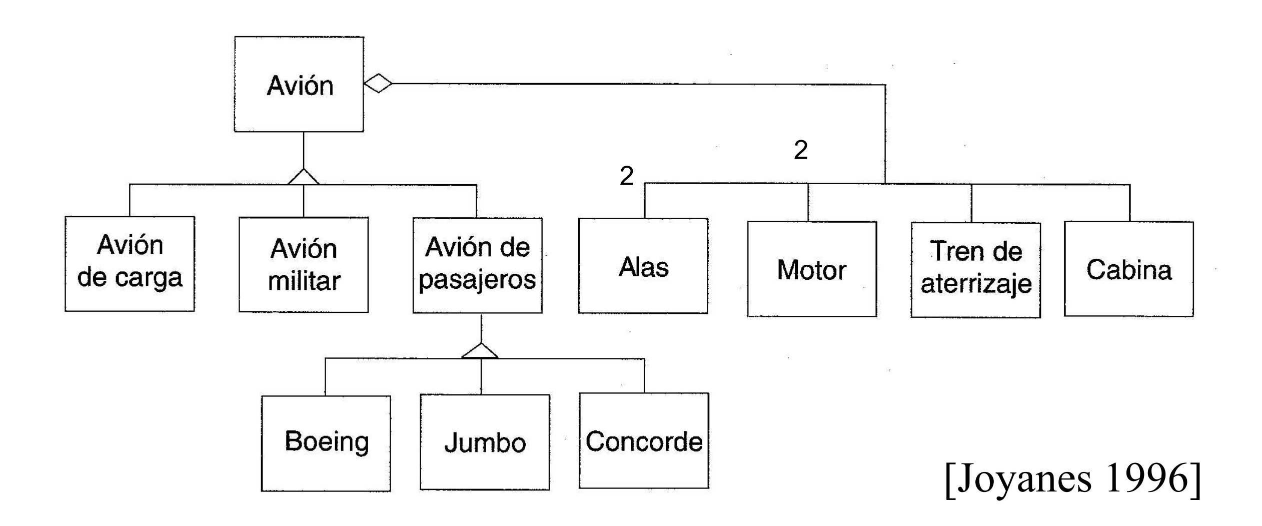
empresario empleados

Cardinalidad

Roles

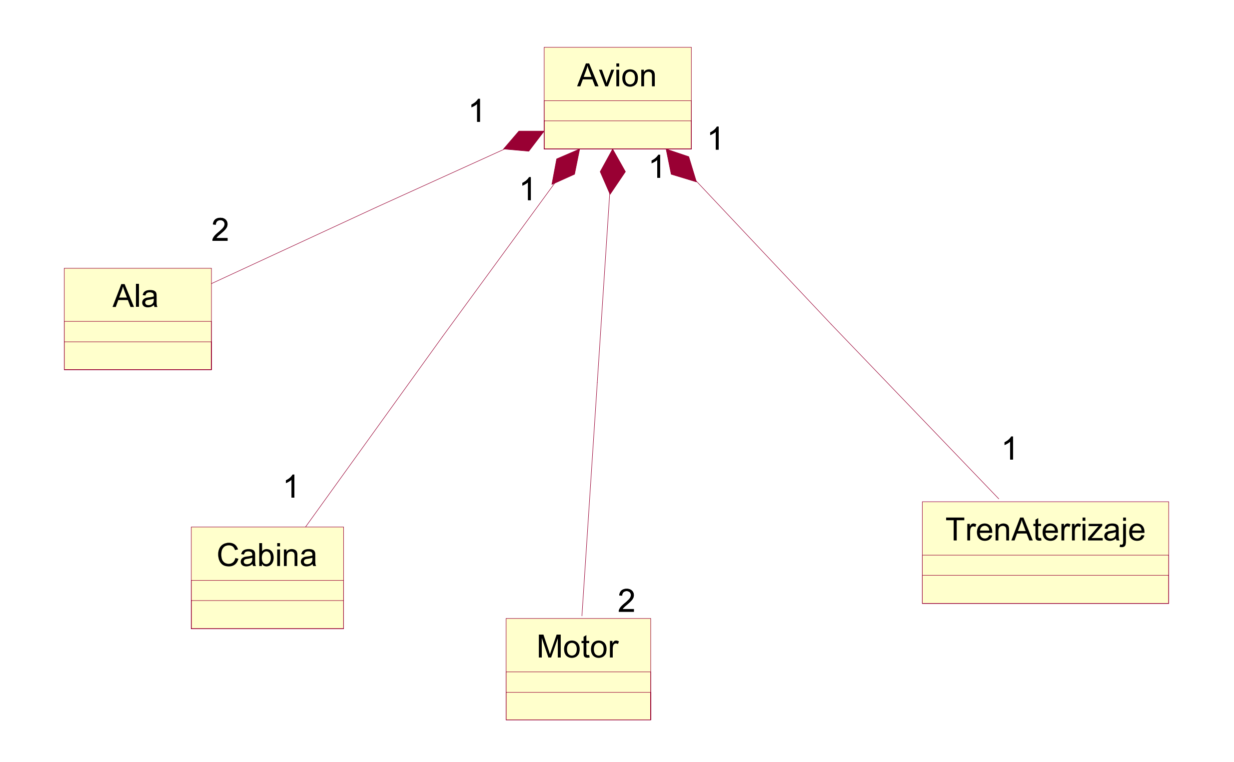
Agregación versus herencia

- Herencia: “*es-un*”
- Agregación: “*parte-de*” o “*tiene-un*”




```
// Implementación en C++
class Ala {...}; // Los ...indican que se ha implementado la clase
class Motor {...};
class TrenDeAterrizaje{...};
class Cabina{...};
class Avion {
          Ala *a1, *a2;
          Motor *m1, *m2;
          TrenAterrizaje *t;
          Cabina *c;
          ...};
```

Composición




```
// Implementación en C++
class Ala {...}; // Los ...indican que se ha implementado la clase
class Motor {...};
class TrenDeAterrizaje{...};
class Cabina{...};
class Avion {
          Ala a1, a2;
          Motor m1, m2;
          TrenDeAterrizaje t;
          Cabina c;
          ...};
```

Clases auxiliares para los ejemplos de agregación y composición en C++

ElementosAvion.h (I)

```
// Módulo ElementosAvion.h (archivo cabecera)
// Clases: Ala, Cabina, Motor, TrenAterrizaje
// versión 2.0, 25 - Enero - 2002
// autor: J.M. Cueva Lovelle
#ifndef ELEMENTOS_AVION_HPP
#define ELEMENTOS_AVION_HPP
#include <iostream>
#include <string>     //Para manejo de string
using namespace std;  //Para usar la biblioteca estándar
class Ala
{
protected:
  string codigo;
  float envergadura;
public:
  Ala(void){codigo="Sin Codigo";envergadura=-1;
            cout<<"Se ha construido el Ala:"<<verCodigo()<<endl;};
  Ala(string unCodigo, float unaEnvergadura)
            {codigo=unCodigo;envergadura=unaEnvergadura;
             cout<<"Se ha construido el Ala:"<<verCodigo()<<endl;};
  void ponCodigo(string unCodigo) {codigo=unCodigo;};
  string verCodigo(void) const {return codigo;};
  void ponEnvergadura(float unaEnvergadura){envergadura=unaEnvergadura;};
  float verEnvergadura(void)const {return envergadura;};
  void verTodo(void)
          {cout<<"Ala:"<<verCodigo()<<" Envergadura:";
           cout<<verEnvergadura()<<endl;};
  ~Ala(void)
    {cout<<"Se ha destruido el Ala:"<<verCodigo()<<endl;};
};
```

Clases auxiliares para los ejemplos de agregación y composición en C++

ElementosAvion.h (II)

```
class Cabina
{
protected:
  string codigo;
  float capacidad;
public:
  Cabina(void){codigo="Sin Codigo";capacidad=-1;
            cout<<"Se ha construido la Cabina:"<<verCodigo()<<endl;};
  Cabina(string unCodigo, float unaCapacidad)
           {codigo=unCodigo;capacidad=unaCapacidad;
            cout<<"Se ha construido la cabina:"<<verCodigo()<<endl;};
  void ponCodigo(string unCodigo) {codigo=unCodigo;};
  string verCodigo(void) const {return codigo;};
  void ponCapacidad(float unaCapacidad){capacidad=unaCapacidad;};
  float verCapacidad(void)const {return capacidad;};
  void verTodo(void)
          {cout<<"Cabina:"<<verCodigo()<<" Capacidad:";
           cout<<verCapacidad()<<endl;};
  ~Cabina(void)
    {cout<<"Se ha destruido la cabina:"<<verCodigo()<<endl;};
};
```

Clases auxiliares para los ejemplos de agregación y composición en C++

ElementosAvion.h (III)

```
class Motor
{
protected:
  string codigo;
  int CV;
public:
  Motor(void){codigo="Sin Codigo";CV=-1;
              cout<<"Se ha construido el Motor:"<<verCodigo()<<endl;};
  Motor(string unCodigo, int unCV){codigo=unCodigo;CV=unCV;
              cout<<"Se ha construido el Motor:"<<verCodigo()<<endl;};
  void ponCodigo(string unCodigo) {codigo=unCodigo;};
  string verCodigo(void) const {return codigo;};
  void ponCV(int unCV){CV=unCV;};
  int verCV(void)const {return CV;};
  void verTodo(void)
          {cout<<"Cabina:"<<verCodigo()<<" CV:";
           cout<<verCV()<<endl;};
  ~Motor(void)
    {cout<<"Se ha destruido el motor:"<<verCodigo()<<endl;};
};
```

Clases auxiliares para los ejemplos de agregación y composición en C++

ElementosAvion.h (IV)

```
class TrenAterrizaje
{
protected:
  string codigo;
  int numRuedas;
public:
  TrenAterrizaje(void)
    {codigo="Sin Codigo";numRuedas=-1;
     cout<<"Se ha construido el tren de aterrizaje:"<<verCodigo()<<endl;};
  TrenAterrizaje(string unCodigo, int unNumRuedas)
    {codigo=unCodigo;numRuedas=unNumRuedas;
     cout<<"Se ha construido el tren de aterrizaje:"<<verCodigo()<<endl;};
  void ponCodigo(string unCodigo) {codigo=unCodigo;};
  string verCodigo(void) const {return codigo;};
  void ponNumRuedas(int unNumRuedas){numRuedas=unNumRuedas;};
  int verNumRuedas(void)const {return numRuedas;};
  void verTodo(void)
          {cout<<"Tren de Aterrizaje :"<<verCodigo()<<" num. ruedas:";
           cout<<verNumRuedas()<<endl;};
  ~TrenAterrizaje(void)
    {cout<<"Se ha destruido el tren de aterrizaje:"<<verCodigo()<<endl;};
};

#endif  //fin ELEMENTOS_AVION_HPP
```

Agregación. Ejemplo en C++ (I) [Stroustrup2000, 24.3.3]

```
// Módulo AvionAgregado.h (archivo cabecera)
// Clase Avion agregando Ala, Cabina, Motor, TrenAterrizaje
// Versión 2.1, 22 - Enero - 2003
// Autor: J.M. Cueva Lovelle
#ifndef AVION_AGREGADO_HPP
#define AVION_AGREGADO_HPP
#include <iostream>
#include <string>     //Para manejo de string
using namespace std;  //Para usar la biblioteca estándar
#include "ElementosAvion.h" //Contiene las clases Ala, Cabina, Motor,...
class Avion
{
 protected:
  Ala *a1, *a2;
  Motor *m1, *m2;
  TrenAterrizaje *t;
  Cabina *c;
  string matricula;
 public:
  Avion(void);
  Avion(string unCodigoAla1, float unaEnvergaduraAla1,
        string unCodigoAla2, float unaEnvergaduraAla2,
        string unCodigoMotor1, int unosCV1,
        string unCodigoMotor2, int unosCV2,
        string unCodigoCabina, float unaCapacidad,
        string unCodigoTrenAterrizaje, int unNumRuedas,
        string unaMatricula);
 Avion (Ala *unA1, Ala *unA2,Motor *unM1, Motor *unM2,TrenAterrizaje *unT,
           Cabina *unC,string unaMatricula);
  void ponMatricula(string unaMatricula) {matricula=unaMatricula;};
  string verMatricula(void) const {return matricula;};
  void verTodo(void);
  ~Avion(void);
};
#endif  //fin AVION_AGREGADO_HPP
```

Agregación. Ejemplo en C++ (II)

```
// Módulo AvionAgregado.cpp
// Implementación  de la Clase Avion
// versión 2.1, 22 - Enero - 2003
// autor: J.M. Cueva Lovelle
#include "AvionAgregado.h"
Avion::Avion(void):a1(new Ala()),a2(new Ala()),
                   m1 (new Motor()), m2(new Motor()),
                   t (new TrenAterrizaje()),
                   c (new Cabina())
{ matricula="Sin Matricula";
  cout<<"Se ha creado el avion de matricula:"<<verMatricula()<<endl; };
Avion::Avion(string unCodigoAla1, float unaEnvergaduraAla1,
             string unCodigoAla2, float unaEnvergaduraAla2,
             string unCodigoMotor1, int unosCV1,
             string unCodigoMotor2, int unosCV2,
             string unCodigoCabina, float unaCapacidad,
             string unCodigoTrenAterrizaje, int unNumRuedas,
             string unaMatricula)
             :a1(new Ala(unCodigoAla1,unaEnvergaduraAla1)),
              a2(new Ala(unCodigoAla2,unaEnvergaduraAla2)),
              m1 (new Motor(unCodigoMotor1,unosCV1)),
              m2(new Motor(unCodigoMotor2,unosCV2)),
              t (new TrenAterrizaje(unCodigoTrenAterrizaje,unNumRuedas)),
              c (new Cabina(unCodigoCabina, unaCapacidad))
          {matricula=unaMatricula; };
Avion::Avion(Ala *unA1, Ala *unA2,
         Motor *unM1, Motor *unM2,
         TrenAterrizaje *unT, Cabina *unC,
         string unaMatricula)
         {
         a1=unA1; a2=unA2;
         m1=unM1; m2=unM2;
         t=unT; c=unC;
         matricula=unaMatricula;
         }
```

Agregación. Ejemplo en C++ (III)

```
void Avion::verTodo(void)
          {cout<<"Elementos del avion de Matricula:";
           cout<<verMatricula()<<endl;
           a1->verTodo(); a2->verTodo();
           m1->verTodo(); m2->verTodo();
           t->verTodo(); c->verTodo(); };
Avion::~Avion(void)
     {delete a1; delete a2; delete m1; delete m2; delete c; delete t;
      cout<<"Se ha destruido el avion de matricula:"<<verMatricula()<<endl; };
```

- Aquí el autor decide destruir los agregados en el destructor de la clase contenedora (Avion), pero podría no ser así. Si se utilizase solamente el constructor donde se pasan los punteros a los objetos, sería más razonable, que se destruyeran externamente a la clase contenedora.

Agregación. Ejemplo en C++ (IV)

Prueba unitaria en C++Builder

```
// Prueba unitaria de la clase Avion (Agregacion.cpp)
// Clase Avion agregando Ala, Cabina, Motor, TrenAterrizaje
// versión 2.1, 23 - Enero - 2003
// autor: J.M. Cueva Lovelle
#pragma hdrstop
#include <condefs.h>
#include <conio>
#include "AvionAgregado.h"
//---------------------------------------------------------------------------
USEUNIT("AvionAgregado.cpp");
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
Avion picosEuropa;
picosEuropa.verTodo();
Avion colloto("AlaIzda",100,"AlaDcha",100,
              "m1",500,"m2",500,
              "c",1000,
              "t",16,
              "COLLOTO-ZPAF");
colloto.verTodo();
Ala *ai=new Ala("AlaI",200);
Ala *ad=new Ala("AlaD",200);
Motor *mRollsI=new Motor("RollsI",1000);
Motor *mRollsD=new Motor("RollsD",1000);
TrenAterrizaje *tMichelin=new TrenAterrizaje("Michelin",16);
Cabina *cAirBus500=new Cabina("CAirBus500",250);
Avion carbayonia (ai,ad,mRollsI,mRollsD,tMichelin,cAirBus500,
                  "CARBAYONIA-GABI");
carbayonia.verTodo();
getch();
return 0;
}
```

Agregación. Ejemplo en C++ (V)
Prueba unitaria en gcc (www.gnu.org)

```
// Prueba unitaria de la clase Avion (Agregado.cpp)
// Clase Avion agregando Ala, Cabina, Motor, TrenAterrizaje
// versión 2.1, 23 - Enero - 2003
// autor: J.M. Cueva Lovelle
// Compilado con gcc (www.gnu.org) en linux de la siguiente forma
// $ g++ -o Agregado.out Agregado.cpp AvionAgregado.cpp

#include "AvionAgregado.h"
int main()
{
 Avion picosEuropa;
 picosEuropa.verTodo();
 Avion colloto("AlaIzda",100,"AlaDcha",100,
              "m1",500,"m2",500,
              "c",1000,
              "t",16,
              "COLLOTO-ZPAF");
colloto.verTodo();
Ala *ai=new Ala("AlaI",200);
Ala *ad=new Ala("AlaD",200);
Motor *mRollsI=new Motor("RollsI",1000);
Motor *mRollsD=new Motor("RollsD",1000);
TrenAterrizaje *tMichelin=new TrenAterrizaje("Michelin",16);
Cabina *cAirBus500=new Cabina("CAirBus500",250);
Avion carbayonia (ai,ad,mRollsI,mRollsD,tMichelin,cAirBus500,
                  "CARBAYONIA-GABI");
carbayonia.verTodo();
 return 0;
}
```

Agregación. Ejemplo en C++ (VI)

Ejecución de la prueba unitaria

```
Se ha construido el Ala:Sin Codigo
Se ha construido el Ala:Sin Codigo
Se ha construido el Motor:Sin Codigo
Se ha construido el Motor:Sin Codigo
Se ha construido el tren de aterrizaje:Sin Codigo
Se ha construido la Cabina:Sin Codigo
Se ha creado el avion de matricula:Sin Matricula
Elementos del avion de Matricula:Sin Matricula
Ala:Sin Codigo Envergadura:-1
Ala:Sin Codigo Envergadura:-1
Cabina:Sin Codigo CV:-1
Cabina:Sin Codigo CV:-1
Tren de Aterrizaje :Sin Codigo num. ruedas:-1
Cabina:Sin Codigo Capacidad:-1
Se ha construido el Ala:AlaIzda
Se ha construido el Ala:AlaDcha
Se ha construido el Motor:m1
Se ha construido el Motor:m2
Se ha construido el tren de aterrizaje:t
Se ha construido la cabina:c
Elementos del avion de Matricula:COLLOTO-ZPAF
Ala:AlaIzda Envergadura:100
Ala:AlaDcha Envergadura:100
Cabina:m1 CV:500
Cabina:m2 CV:500
Tren de Aterrizaje :c num. ruedas:1000
Cabina:t Capacidad:16
```

Agregación. Ejemplo en C++ (VII)

Ejecución de la prueba unitaria (continuación)

```
Se ha construido el Ala:AlaI
Se ha construido el Ala:AlaD
Se ha construido el Motor:RollsI
Se ha construido el Motor:RollsD
Se ha construido el tren de aterrizaje:Michelin
Se ha construido la cabina:CAirBus500
Elementos del avion de Matricula:CARBAYONIA-GABI
Ala:AlaI Envergadura:200
Ala:AlaD Envergadura:200
Cabina:RollsI CV:1000
Cabina:RollsD CV:1000
Tren de Aterrizaje :Michelin num. ruedas:16
Cabina:CAirBus500 Capacidad:250
Se ha destruido el Ala:AlaI
Se ha destruido el Ala:AlaD
Se ha destruido el motor:RollsI
Se ha destruido el motor:RollsD
Se ha destruido la cabina:CAirBus500
Se ha destruido el tren de aterrizaje:Michelin
Se ha destruido el avion de matricula:CARBAYONIA-GABI
Se ha destruido el Ala:AlaIzda
Se ha destruido el Ala:AlaDcha
Se ha destruido el motor:m1
Se ha destruido el motor:m2
Se ha destruido la cabina:c
Se ha destruido el tren de aterrizaje:t
Se ha destruido el avion de matricula:COLLOTO-ZPAF
Se ha destruido el Ala:Sin Codigo
Se ha destruido el Ala:Sin Codigo
Se ha destruido el motor:Sin Codigo
Se ha destruido el motor:Sin Codigo
Se ha destruido la cabina:Sin Codigo
Se ha destruido el tren de aterrizaje:Sin Codigo
Se ha destruido el avion de matricula:Sin Matricula
```

Composición. Ejemplo en C++ (I) [Stroustrup1997, 24.3.3]

```
// Módulo AvionCompuesto.h (archivo cabecera)
// Clase Avion por composicion de Ala, Cabina, Motor, TrenAterrizaje
// versión 2.0, 25 - Enero - 2002
// autor: J.M. Cueva Lovelle
#ifndef AVION_COMPUESTO_HPP
#define AVION_COMPUESTO_HPP
#include <iostream>
#include <string>     //Para manejo de string
using namespace std;  //Para usar la biblioteca estándar
#include "ElementosAvion.h"
class Avion
{
 protected:
  Ala a1, a2;
  Motor m1, m2;
  TrenAterrizaje t;
  Cabina c;
  string matricula;
 public:
  Avion(void);
  Avion(string unCodigoAla1, float unaEnvergaduraAla1,
        string unCodigoAla2, float unaEnvergaduraAla2,
        string unCodigoMotor1, int unosCV1,
        string unCodigoMotor2, int unosCV2,
        string unCodigoCabina, float unaCapacidad,
        string unCodigoTrenAterrizaje, int unNumRuedas,
        string unaMatricula);
  void ponMatricula(string unaMatricula) {matricula=unaMatricula;};
  string verMatricula(void) const {return matricula;};
  void verTodo(void);
  ~Avion(void);
};
#endif  //fin AVION_COMPUESTO_HPP
```

Composición. Ejemplo en C++ (II)

```
// Módulo AvionCompuesto.cpp
// Implementación  de la Clase AvionCompuesto
// versión 2.0, 25 - Enero - 2002
// autor: J.M. Cueva Lovelle
#include "AvionCompuesto.h"
Avion::Avion(void):a1(),a2(),m1 (), m2(),t (),c ()
{ matricula="Sin Matricula";
  cout<<"Se ha creado el avion de matricula:"<<verMatricula()<<endl; };
Avion::Avion(string unCodigoAla1, float unaEnvergaduraAla1,
             string unCodigoAla2, float unaEnvergaduraAla2,
             string unCodigoMotor1, int unosCV1,
             string unCodigoMotor2, int unosCV2,
             string unCodigoCabina, float unaCapacidad,
             string unCodigoTrenAterrizaje, int unNumRuedas,
             string unaMatricula)
             :a1(unCodigoAla1,unaEnvergaduraAla1),
              a2(unCodigoAla2,unaEnvergaduraAla2),
              m1 (unCodigoMotor1,unosCV1),
              m2(unCodigoMotor2,unosCV2),
              t (unCodigoTrenAterrizaje,unNumRuedas),
              c (unCodigoCabina, unaCapacidad)
{ matricula=unaMatricula;
 cout<<"Se ha creado el avion de matricula:";
 cout<<verMatricula()<<endl; };
void Avion::verTodo(void)
          { cout<<"Elementos del avion de Matricula:";
            cout<<verMatricula()<<endl;
            a1.verTodo(); a2.verTodo();
            m1.verTodo(); m2.verTodo();
            t.verTodo(); c.verTodo(); };
Avion::~Avion(void)
        {cout<<"Se ha destruido el avion de matricula:";
         cout<<verMatricula()<<endl; };
```

Composición. Ejemplo en C++ (III)

Prueba unitaria de la clase Avion con C++Builder

```
// Prueba unitaria de la clase Avion (Composicion.cpp)
// Clase Avion componiendo Ala, Cabina, Motor, TrenAterrizaje
// versión 2.0, 25 - Enero - 2002
// autor: J.M. Cueva Lovelle
// Compilado con C++Builder 4.0 en modo consola
#pragma hdrstop
#include <condefs.h>
#include <conio>
#include "AvionCompuesto.h"
//---------------------------------------------------------------------------
USEUNIT("AvionCompuesto.cpp");
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
 Avion picosEuropa;

 picosEuropa.verTodo();

 Avion colloto("AlaIzda",100,"AlaDcha",100,
              "m1",500,"m2",500,
              "c",1000,
              "t",16,
              "COLLOTO-ZPAF");

 colloto.verTodo();
        getch();
        return 0;
}
```

Composición. Ejemplo en C++ (IV)

Prueba unitaria de la clase Avion utilizando gcc (www.gnu.org)

```
// Prueba unitaria de la clase Avion (Composicion.cpp)
// Clase Avion componiendo Ala, Cabina, Motor, TrenAterrizaje
// versión 2.0, 25 - Enero - 2002
// autor: J.M. Cueva Lovelle
// Compilado con gcc de www.gnu.org en linux
// $ g++ -o Composicion.out Composicion.cpp AvionCompuesto.cpp

#include "AvionCompuesto.h"

int main( )
{
 Avion picosEuropa;

 picosEuropa.verTodo();

 Avion colloto("AlaIzda",100,"AlaDcha",100,
              "m1",500,"m2",500,
              "c",1000,
              "t",16,
              "COLLOTO-ZPAF");

 colloto.verTodo();

  return 0;
}
```

Composición. Ejemplo en C++ (V)

Ejecución de la prueba unitaria de la clase Avion

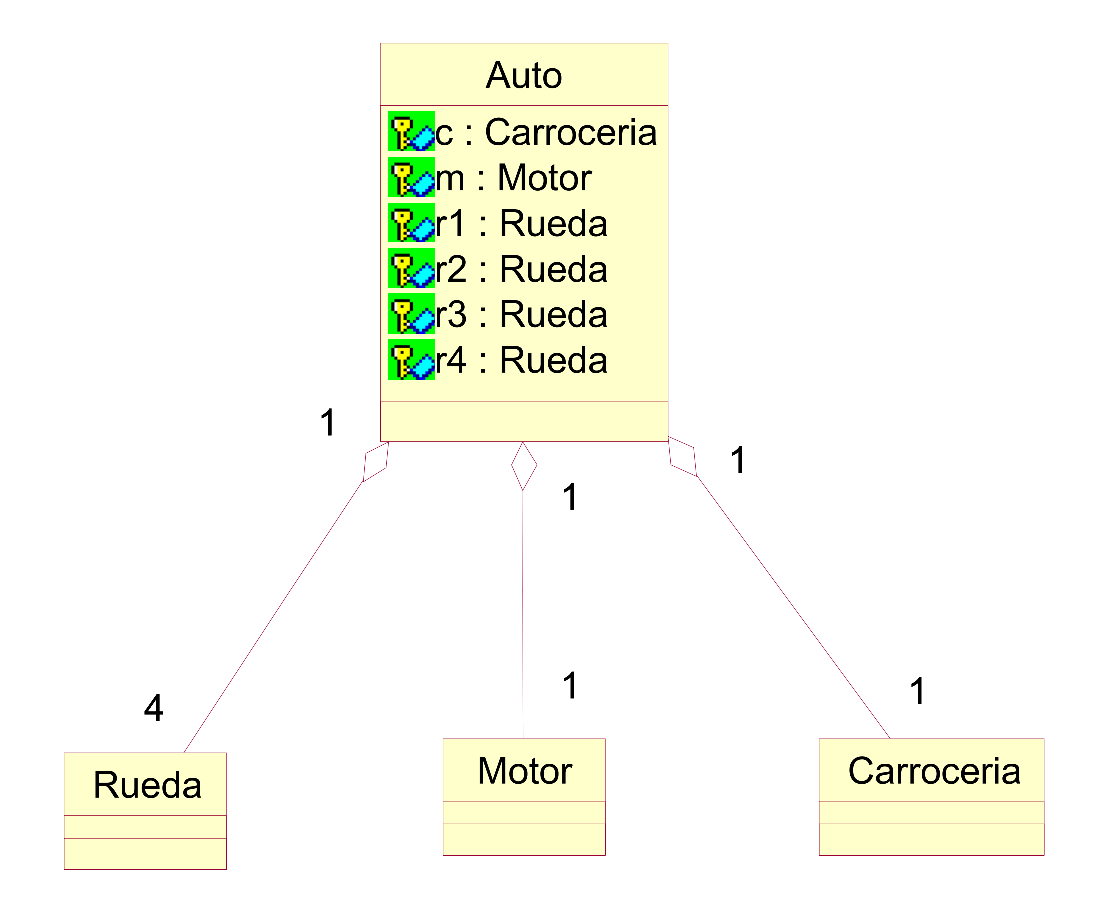
```
Se ha construido el Ala:Sin Codigo
Se ha construido el Ala:Sin Codigo
Se ha construido el Motor:Sin Codigo
Se ha construido el Motor:Sin Codigo
Se ha construido el tren de aterrizaje:Sin Codigo
Se ha construido la Cabina:Sin Codigo
Se ha creado el avion de matricula:Sin Matricula
Elementos del avion de Matricula:Sin Matricula
Ala:Sin Codigo Envergadura:-1
Ala:Sin Codigo Envergadura:-1
Cabina:Sin Codigo CV:-1
Cabina:Sin Codigo CV:-1
Tren de Aterrizaje :Sin Codigo num. ruedas:-1
Cabina:Sin Codigo Capacidad:-1
Se ha construido el Ala:AlaIzda
Se ha construido el Ala:AlaDcha
Se ha construido el Motor:m1
Se ha construido el Motor:m2
Se ha construido el tren de aterrizaje:t
Se ha construido la cabina:c
Se ha creado el avion de matricula:COLLOTO-ZPAF
Elementos del avion de Matricula:COLLOTO-ZPAF
Ala:AlaIzda Envergadura:100
Ala:AlaDcha Envergadura:100
Cabina:m1 CV:500
Cabina:m2 CV:500
Tren de Aterrizaje :t num. ruedas:16
Cabina:c Capacidad:1000
```

Composición. Ejemplo en C++ (V)
Ejecución de la prueba unitaria de la clase Avion (continuación)

```
Se ha destruido el avion de matricula:COLLOTO-ZPAF
Se ha destruido la cabina:c
Se ha destruido el tren de aterrizaje:t
Se ha destruido el motor:m2
Se ha destruido el motor:m1
Se ha destruido el Ala:AlaDcha
Se ha destruido el Ala:AlaIzda
Se ha destruido el avion de matricula:Sin Matricula
Se ha destruido la cabina:Sin Codigo
Se ha destruido el tren de aterrizaje:Sin Codigo
Se ha destruido el motor:Sin Codigo
Se ha destruido el motor:Sin Codigo
Se ha destruido el Ala:Sin Codigo
Se ha destruido el Ala:Sin Codigo
```

- Puede observarse que no se produce en el mismo orden la destrucción de los objetos que componen el avión según utilicemos Agregación o Composición

Agregación
Ejemplo en Java



Agregación

Una implementación en Java (I)
Clase Carrocería

```
class Carroceria {

 protected String color="Blanco";
 protected int numeroSerie;
 protected static int contador;

 Carroceria(){
  numeroSerie=contador++;
 }

 Carroceria(String unColor){
  this();
  color=unColor;
  System.out.println("Se ha fabricado una carrocería de color "+
         color);
  }

  public String verColor() {
   return color;
  }

  public int verNumeroSerie(){
    return numeroSerie;
  }

  public static int verContador(){
    return contador;
  }

} //Fin de la clase Carrocería
```

Agregación

Una implementación en Java (II)

Construcción de la clase Auto por composición de objetos de las clases Carrocería, Motor y Rueda

```
class Motor {} //Implementación vacía

class Rueda {} //Implementación vacía

public class Auto {

          protected Carroceria c;
          protected Motor m;
          protected Rueda r1,r2,r3,r4;

          Auto(){
                    c = new Carroceria();
                    m = new Motor();
                    r1 = new Rueda();
                    r2 = new Rueda();
                    r3 = new Rueda();
                    r4 = new Rueda();
          }

          Auto(String unColor){
                    c = new Carroceria(unColor);
                    m = new Motor();
                    r1 = new Rueda();
                    r2 = new Rueda();
                    r3 = new Rueda();
                    r4 = new Rueda();
          }

public String toString(){
 String describe = "Auto con carroceria de color "
          + c.verColor()+'\n'
          + "y número de serie " + c.verNumeroSerie();
 return describe;
 }
```

Agregación

Una implementación en Java (III)

Prueba unitaria de la clase Auto

```
public static void main (String[] args) {

  Auto miCoche = new Auto();

  System.out.println("Hola "+ miCoche);

  Auto otroCoche = new Auto("Azul");

  System.out.println("Número de carrocerias producidas "
                      + Carroceria.verContador());

 } // Fin de main

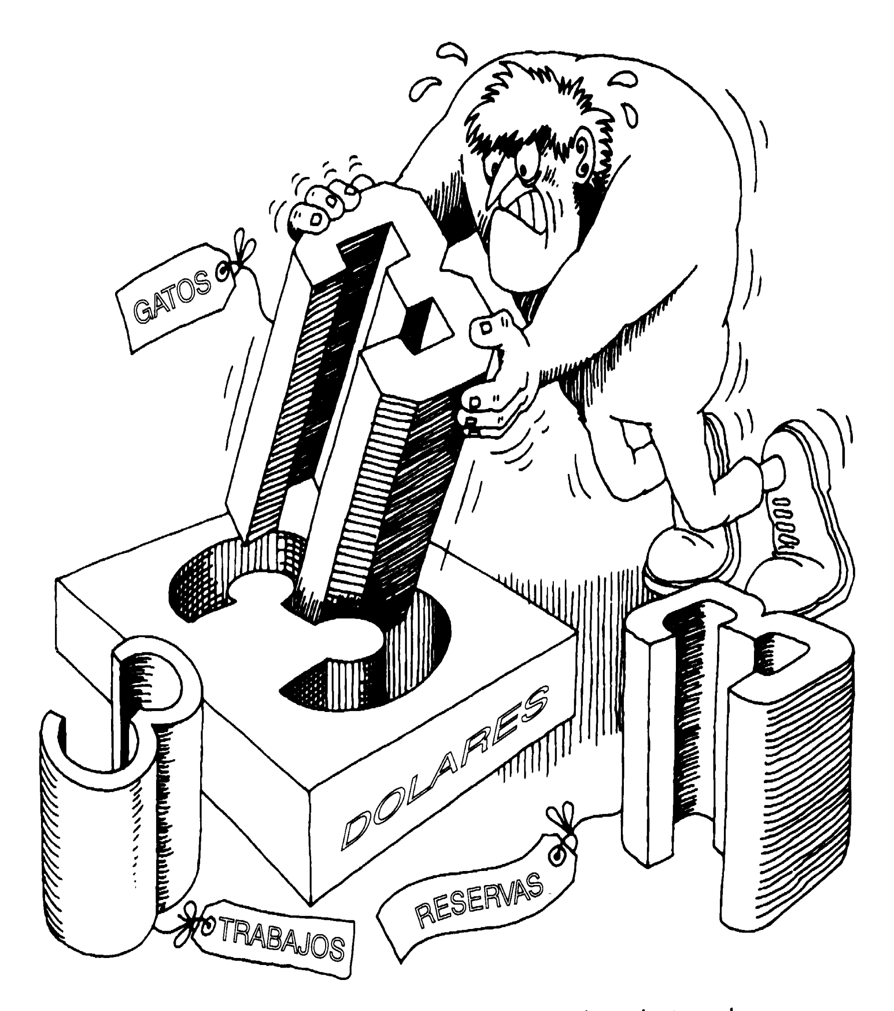
} // Fin de la clase Auto
```

Elementos del modelo de objetos

- **Elementos fundamentales**
  - ✓ Abstracción
  - ✓ Encapsulación y ocultación de la información
  - ✓ Modularidad
  - ✓ Jerarquía (herencia, polimorfismo y agregación)
- **Elementos secundarios**
  - Tipos (control de tipos)
  - Concurrencia
  - Persistencia
  - Distribución
  - Sobrecarga
  - Genericidad
  - Manejo de excepciones
- **Elementos relacionados**
  - Componentes
  - Patrones de diseño

3.8 Control estricto de tipos

Los tipos son la puesta en vigor de la clase de los objetos, de modo que los objetos de tipos distintos no pueden intercambiarse o, como mucho, pueden intercambiarse de formas muy restringidas [Booch 94]



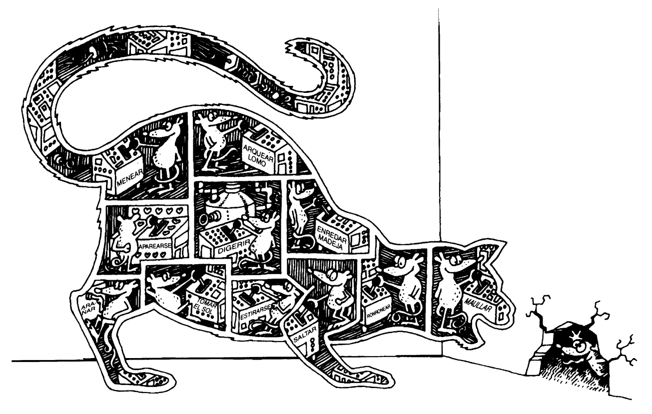
La comprobación estricta de tipos impide que se mezclen abstracciones.

Control estricto de tipos

- Se intercambiará el concepto de tipo y clase, aunque en algunos lenguajes Orientados a Objetos no son exactamente lo mismo
- La congruencia de tipos se produce en la comprobación estricta de forma que las reglas del dominio dictan y refuerzan ciertas combinaciones correctas de las abstracciones
- Los lenguajes orientados a objetos pueden ser
  - con comprobación estricta de tipos
  - con comprobación debil de tipos
  - sin tipos
- Lenguajes con comprobación estricta de tipos (Java, Eiffel, Ada, Object Pascal,C#)
  - La concordancia de tipos se impone de manera estricta
  - No puede llamarse a una operación sobre un objeto a menos que en la clase o superclases del objeto esté definido el prototipo exacto de esa operación
  - La violación de tipos se detecta en tiempo de compilación
  - La comprobación estricta de tipos permite utilizar el lenguaje de programación para imponer ciertas decisiones de diseño, y por eso es particularmente relevante a medida que aumenta la complejidad del sistema
  - El inconveneiente principal es que pequeños cambios en el interfaz de una clase base obliga a la recompilación de todas las subclases
  - Si el lenguaje no soporta genericidad (clases parametrizadas) es problemático manejar colecciones de objetos heterogéneos seguras respecto al tipo. Si el lenguaje si las soporta (C++) se puede utilizar una clase contenedor segura respecto al tipo
  - Una primera solución es utilizar punteros genéricos (void en C y C++, pointer en Pascal y Object Pascal)
  - La segunda solución es la identificación de tipos en tiempo de ejecución (Smalltalk y C++)
  - Una tercera solución es utilizar operaciones polimórficas utilizando ligadura dinámica o tardía
- Lenguajes con comprobación débil o híbrida de tipos (C++)
  - Permiten ignorar o suprimir las reglas de comprobación de tipos
- Lenguajes sin tipos (Smalltalk)
  - Se puede enviar cualquier mensaje a cualquier clase aunque ésta desconozca como responder al mensaje
  - Los errores se detectan en tiempo de ejecución

3.9 Concurrencia

Es la propiedad que distingue un objeto activo de uno que no está activo *[Booch 94]*



La concurrencia permite a diferentes objetos actuar al mismo tiempo.

Concurrencia

- La concurrencia se centra en la abstracción de hilos (threads) y en la sincronización
- Los objetos pueden adquirir estas propiedades
- El diseño orientado a objetos puede construir modelos como un conjunto de objetos cooperativos, algunos de los cuales son activos y sirven así como centros de actividad independiente
- La mayor parte de los sistemas operativos actuales dan soporte a la concurrencia (UNIX, Linux, OS/2, familia Windows: NT, 95, 2000, XP,...)
- La concurrencia es una característica intrínseca de ciertos lenguajes de programación (*Java*)
  - En *Ada* las tareas (*task*)
  - En *Smalltalk* la clase *Process*
- En otros es una característica añadida por medio de bibliotecas, por ejemplo C++ y la biblioteca de tareas de AT&T con las clases Sched, Timer, Task y otras
- En último caso pueden utilizarse interrupciones si el sistema operativo no tiene concurrencia (MS-DOS)

3.10 Persistencia

Es la propiedad de un objeto por la que su existencia transciende el tiempo (es decir, el objeto continúa existiendo después de que su creador deja de existir) y/o el espacio (es decir, la posición del objeto varía con respecto al espacio de direcciones en el que fue creado) *[Booch 94]*

Esfinge 2500 años A.C







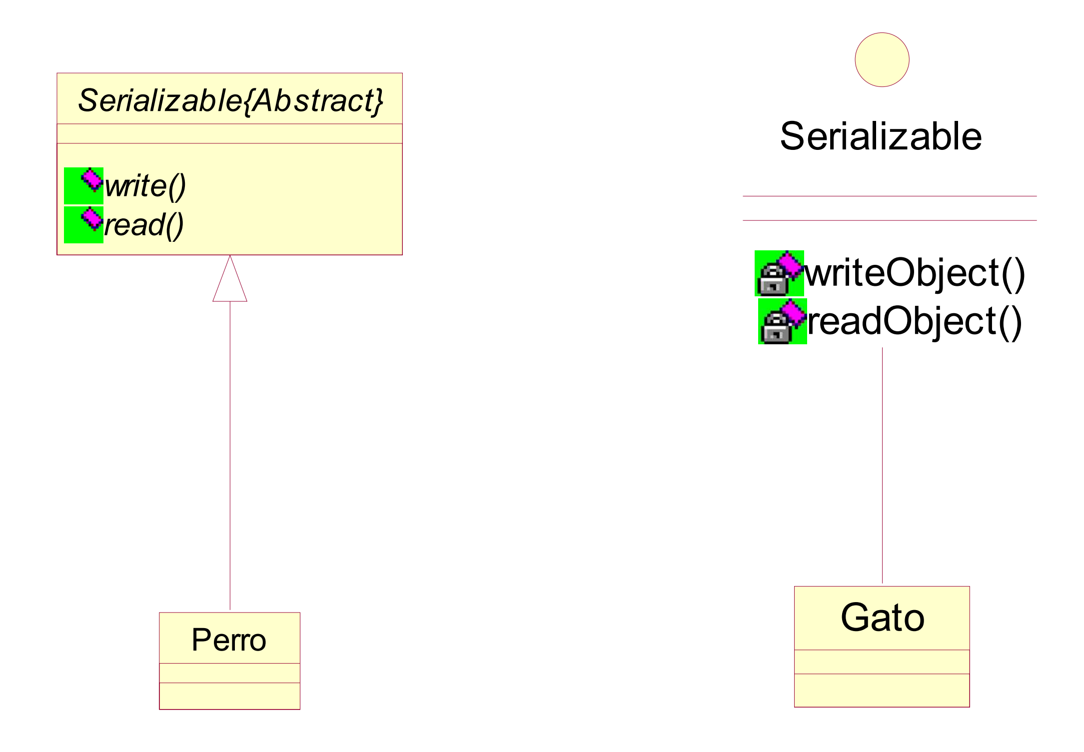
El objeto continúa existiendo después de que su creador deja de existir.

Persistencia

- La persistencia conserva el estado de un objeto en el tiempo y en el espacio *[Booch 94]*
- La persistencia implica más que la mera duración de los datos, no sólo persiste el estado del objeto, sino que su clase debe trascender a cualquier programa individual, de forma que todos los programas interpreten de la misma manera el estado almacenado
- Los lenguajes orientados a objetos ofrecen el soporte de la persistencia por medio de algún artilugio (almacenamiento de clases y datos,…)
- El almacenar objetos en ficheros es una solución ingenua para la persistencia, pues los objetos no son sólo datos
- El principal problema es que los sistemas operativos actuales están basados en ficheros

Serialización

- También denominada persistencia ligera
- La serialización de objetos es un método que permite intercambiar la información de un objeto entre un emisor y un receptor conectados (por ejemplo usando TCP/IP)
- Para serializar un objeto el emisor escribe un flujo de bytes en un `stream` de datos
- El stream de datos contiene información de la clase y de los atributos del objeto serializado
- El receptor lee el `stream` de datos y es capaz de reconstruir el objeto correspondiente.
- Una forma de implementar la serialización en C++ es haciendo que las clases cuyos objetos se vayan a serializar hereden de una clase abstracta Serializable e implementen los métodos `Serializable::read()` y `Serializable::write()`
- En Java es necesario implementar la interfaz `Serializable`, definida en el API del lenguaje



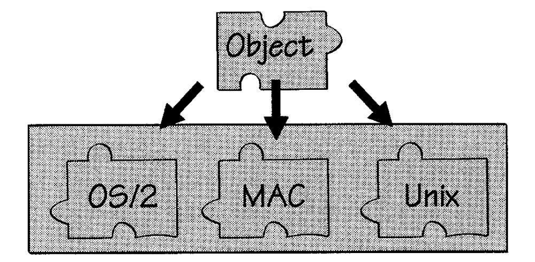
3.11 Distribución

- Computación de objetos distribuidos (*Distributed Object Computing, DOC*)
  - Es un paradigma de computación que distribuye objetos de cooperación a través de una red heterogénea y permite que los objetos interoperen como un todo unificado
- Beneficios de la computación con objetos distribuidos
  - Independencia de la plataforma y sistema operativo (*plug-and-play*)
  - Integración mediante uso de envoltorios (*wrapping*) de aplicaciones heredadas (*legacy applications)*
  - Descomposición del sistema en componentes
  - Flexibilidad para adaptarse a los cambios
  - Simplicidad cuando se distribuyen aplicaciones complejas en entornos heterogéneos

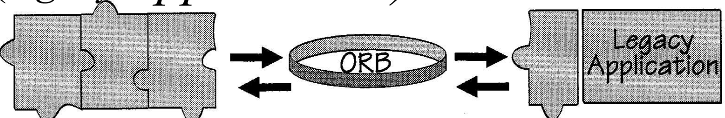
Distribución

Beneficios de los objetos distribuidos [Orfali 1996]

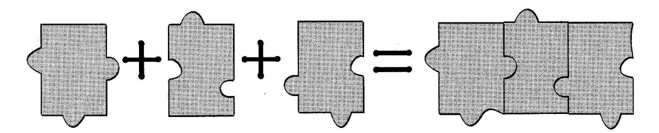
• Independencia de la plataforma y sistema operativo



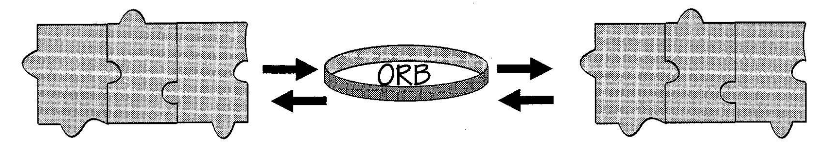
•Integración mediante uso de envoltorios (*wrapping*) de aplicaciones heredadas (*legacy applications)*



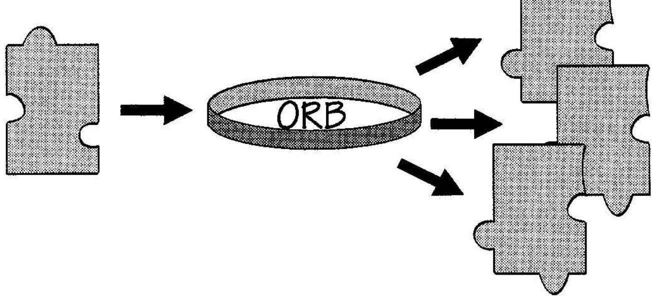
• Descomposición del sistema en componentes (*plug-and-play*)



• Flexibilidad para adaptarse a los cambios



• Simplicidad cuando se distribuyen aplicaciones complejas en entornos heterogéneos



Soluciones para manejar objetos distribuidos

- OMG : **CORBA** URL http://www.omg.org



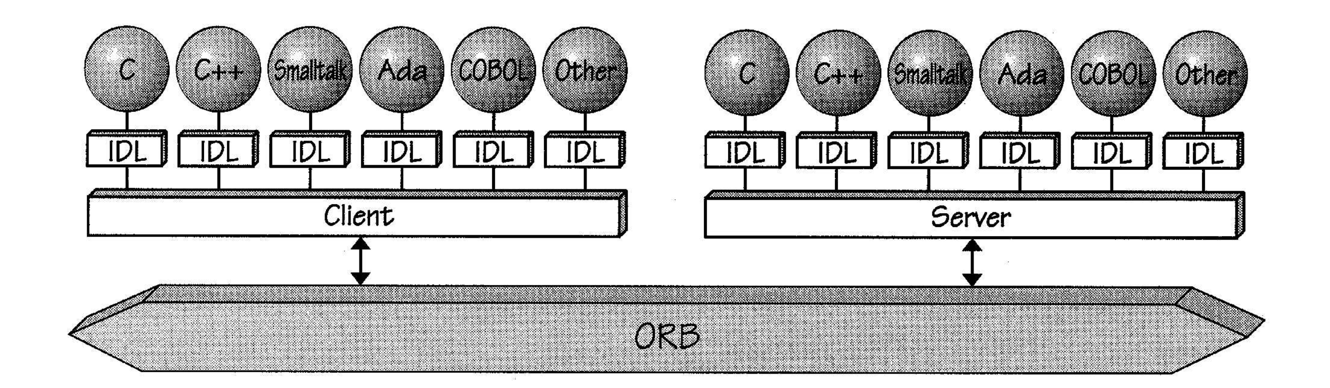
- Microsoft
  - **CORBA**
  - **DCOM :Distributed Component Object Model**
    - Es un protocolo que permite comunicarse a los componentes a través de las redes de los sistemas operativos de la familia windows.
    - Existen puentes DCOM-CORBA
    - http://www.microsoft.com/com/tech/DCOM.asp
  - **.NET Remoting**
- Sun: CORBA y RMI
  - Java™ **RMI**: Remote Method Invocation
    - Permite un acceso fácil a objetos distribuidos de Java
    - http://java.sun.com/products/jdk/rmi/
- Otras soluciones: **Agentes móviles**

Distribución

Terminología



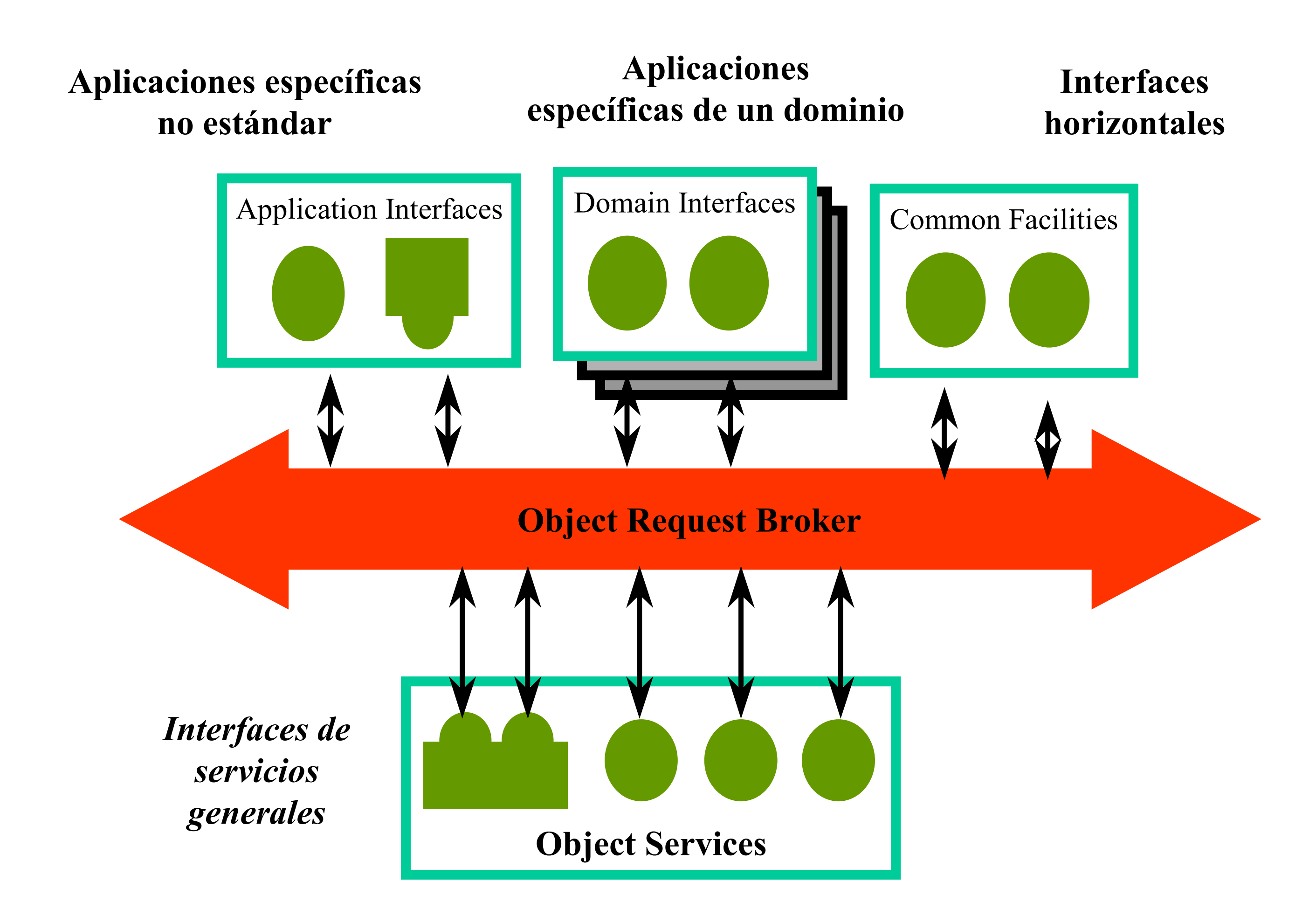
- **OMG**, *Object Management Group.* **URL** http://www.omg.org
  - Consorcio de fabricantes que trabaja desde 1989 para definir estándares en Tecnología de Objetos.
  - Uno de sus estándares es CORBA, que define las especificaciones necesarias para que sobre un bus de software abierto objetos de distintos fabricantes puedan interoperar a través de diferentes redes y sistemas operativos.
  - Otro estándar aprobado por OMG es UML.
- **ORB**, *Object Request Broker.* Gestor de peticiones de objetos
  - Es el mecanismo que permite a los objetos comunicarse con otros a través de la red
- **CORBA**, *Common Object Request Broker Architecture*
  - Especificación de OMG para que distintos ORB puedan trabajar juntos
- **IDL**, *Interface Definition Language*
  - es un lenguaje para definir el interfaz de los componentes, que permite el manejo de objetos distribuidos a través de una red



- Consta de cinco grandes apartados, que también se denomina OMA (Object Management Architecture) a la visión arquitectónica de la norma.
  - Un lenguaje de definición de interfaces (**IDL**, *Interface Definition Languaje)* y una infraestructura de distribución de objetos (**ORB**, *Object Request Broker)*. Este apartado da el nombre a la norma CORBA (Arquitectura Común del Gestor de Peticiones de Objetos)
  - Una descripción de servicios comunes necesarios sistemáticamente para los objetos (**Corba Services**)
  - Una descripción de servicios comunes necesarios sistemáticamente para las aplicaciones, y no ya solamente para los objetos (**Corba Facilities**)
  - Una colección de descripciones de servicios especializados (**Domain Services)**
  - Una especificación normativa de compatibilidad entre ORB (denominada CORBA 2.0)



Object Management Architecture (OMA)



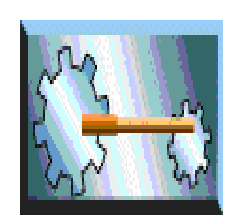
CORBA: La arquitectura Inter-ORB

- GIOP (General Inter-ORB Protocol)
  – Especifica un conjunto de formatos de mensajes y de representaciones comunes de datos para comunicar ORBs
- IIOP (Internet Inter-ORB Protocol)
  – Especifica como se intercambian los mensajes GIOP a través de una red TCP/IP
- ESIOPs (Environment-Specific Inter-ORB Protocols)
  – Especifica como se interopera sobre redes específicas

Implementaciones de CORBA

Orbix	IONA Technologies
NEO	SunSoft
Orb Plus	HP
DSOM	IBM
M3	BEA
VisiBroker	Borland
PowerBroker	ExperSoft
DAIS	ICL
D.O.M.E.	OOT

¿Qué se encuentra en un ORB?



■ Compilador IDL

- Correspondencia entre IDL y un lenguaje objeto (C++, Java, Ada, Smalltalk, ...)
- Sigue el estándar CORBA
- Produce los elementos cliente y servidor

`gridC.cc`

Parte “Stub” cliente

`gridS.cc`

Parte “Skeleton” servidor



■ ORB Runtime

- El Runtime llama al código Stub y Skeleton
- El Runtime permite olvidarse de los protocolos de red
- Generalmente se implementa con una pareja de bibliotecas

ORB client library	ORB server library

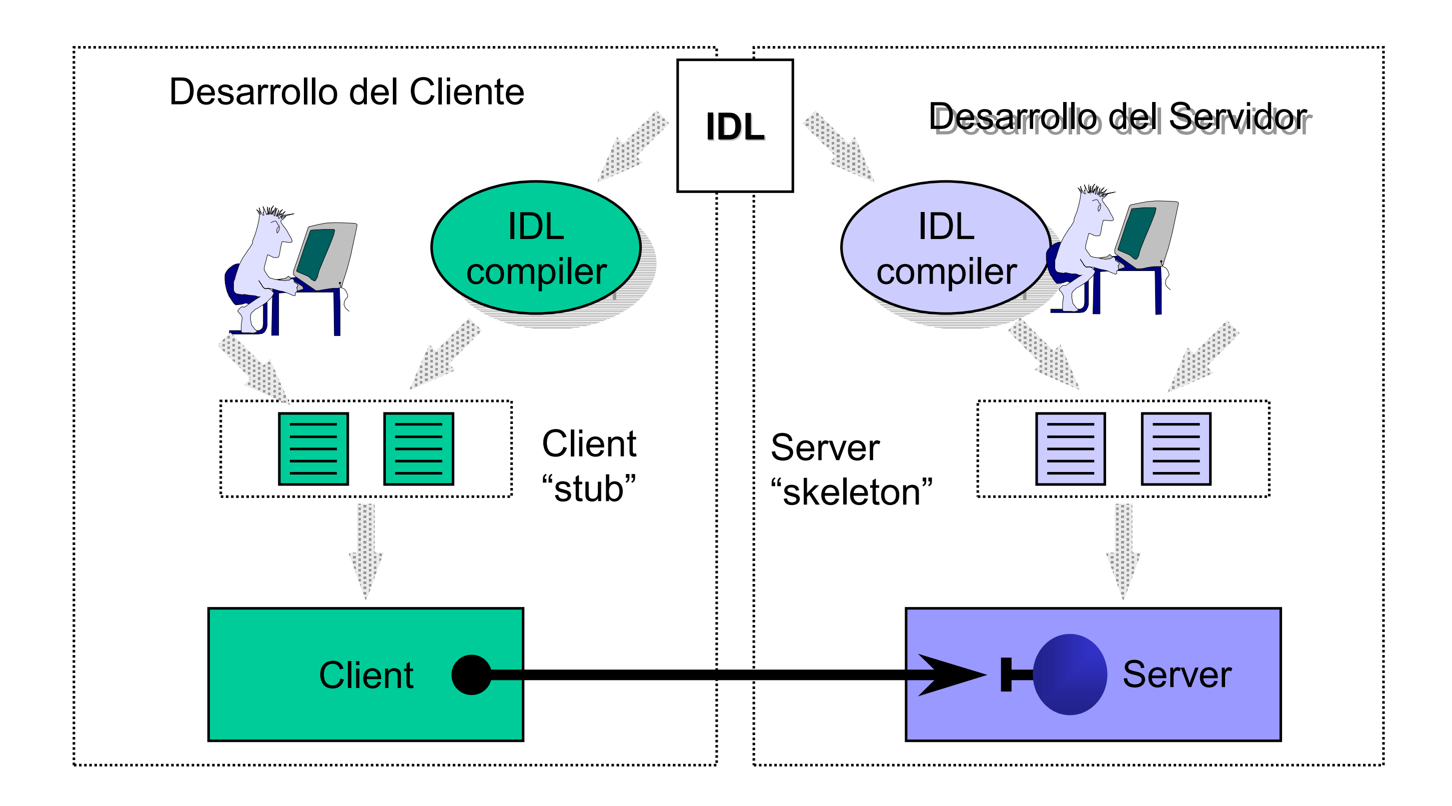


■ Componente Activación

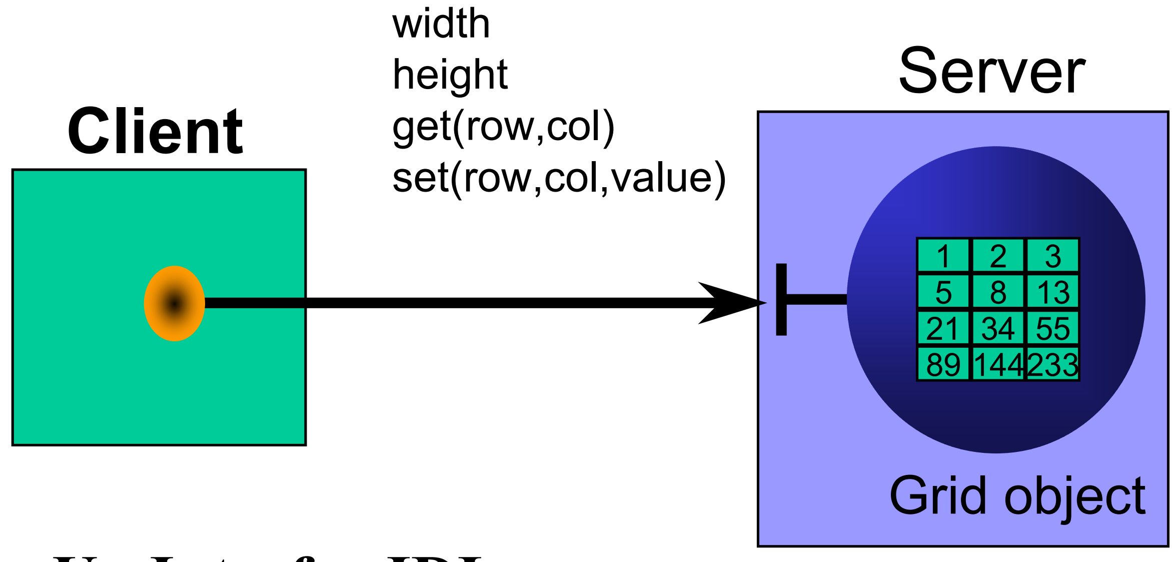
•Tiene dos funciones principales
  •Puede arrancar o activar el servidor
  •Puede llevar a cabo la conexión inicial entre el cliente y el servidor
•Los servidores pueden “activarse” por demanda
  •Lo más típico es que el cliente inicie la demanda
  •Pero existen más opciones
•Generalmente hay uno por host

Pasos para construir una aplicación

- Escribir el IDL
- Compilar el IDL al lenguaje objeto
- Escribir un Cliente
- Implementar la interfaz o Interfaces
- Escribir un Servidor
- Registrar



Ejemplo - Un objeto Grid

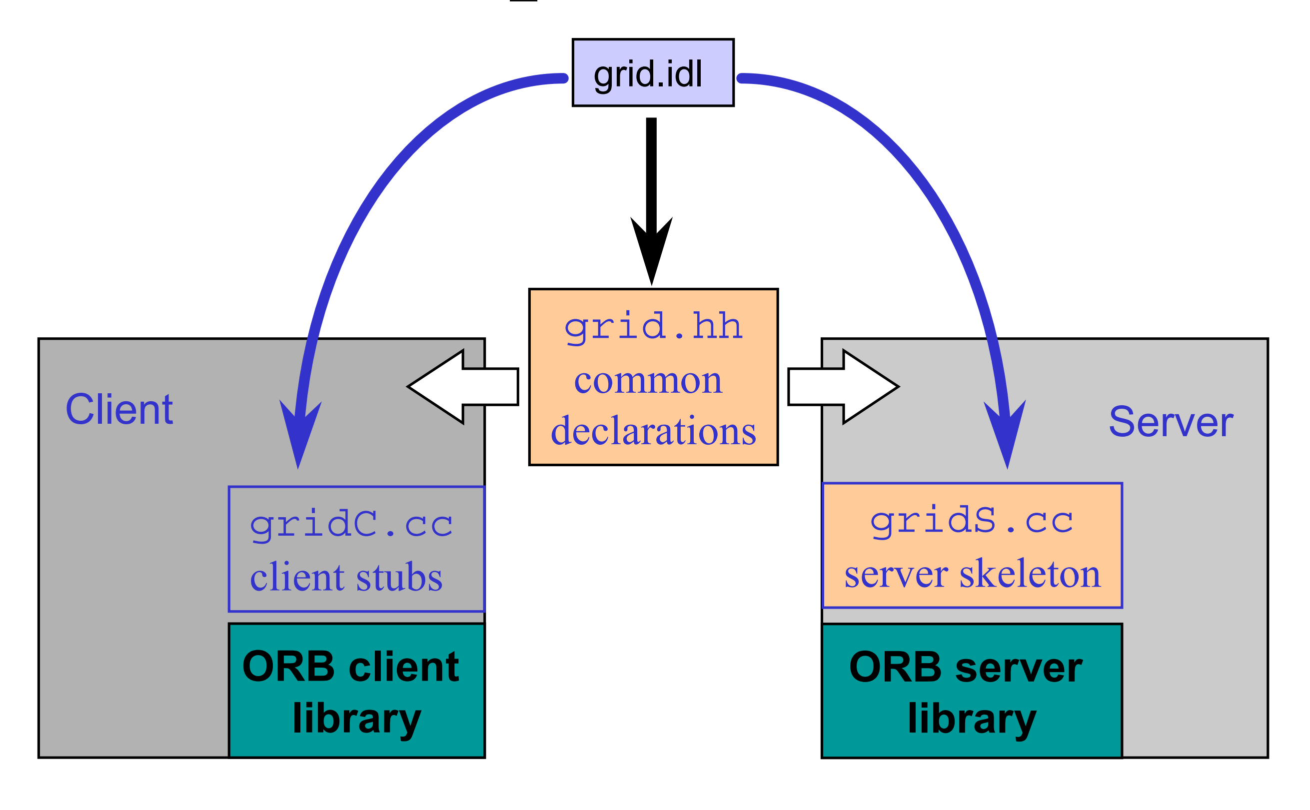


Un Interfaz IDL

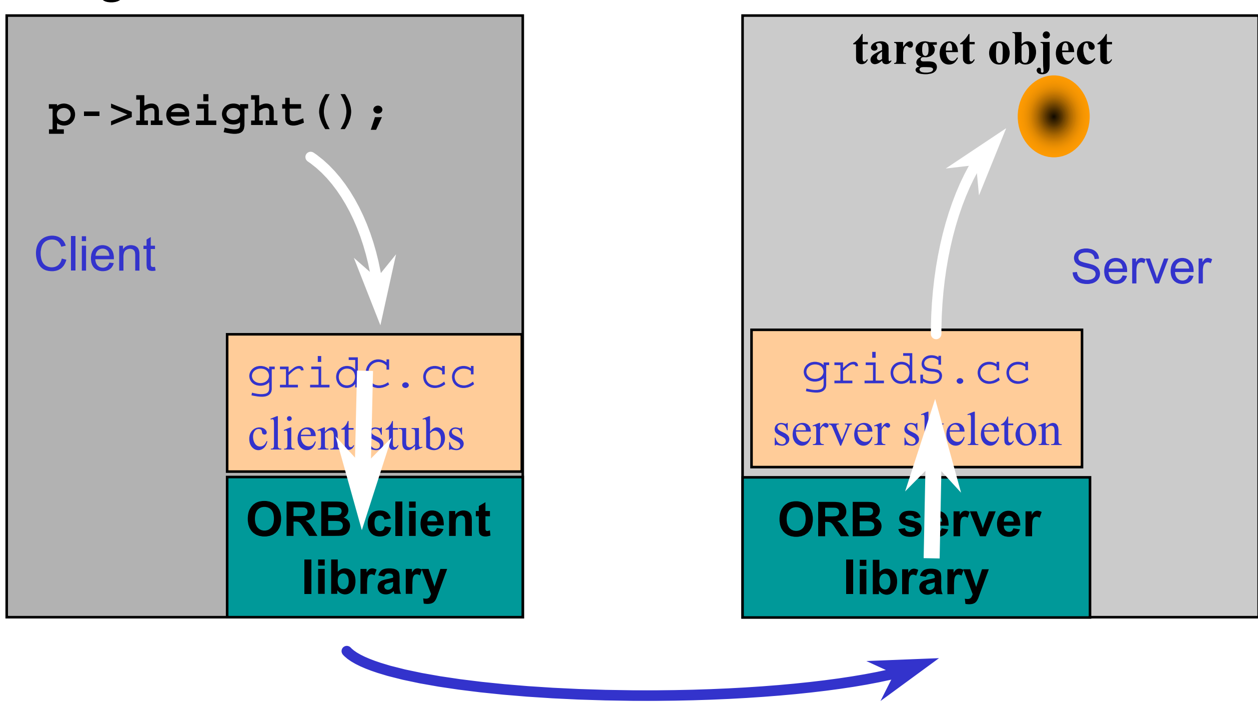
```
interface Grid {
 readonly attribute short height;
 readonly attribute short width;
 void set(in short row, in short col,in long value);
 long get(in short row, in short col);
};
```

grid.idl

Compilando a C++



Ejecución



Ejemplo de código cliente

```
#include "grid.hh"
#include <iostream.h>

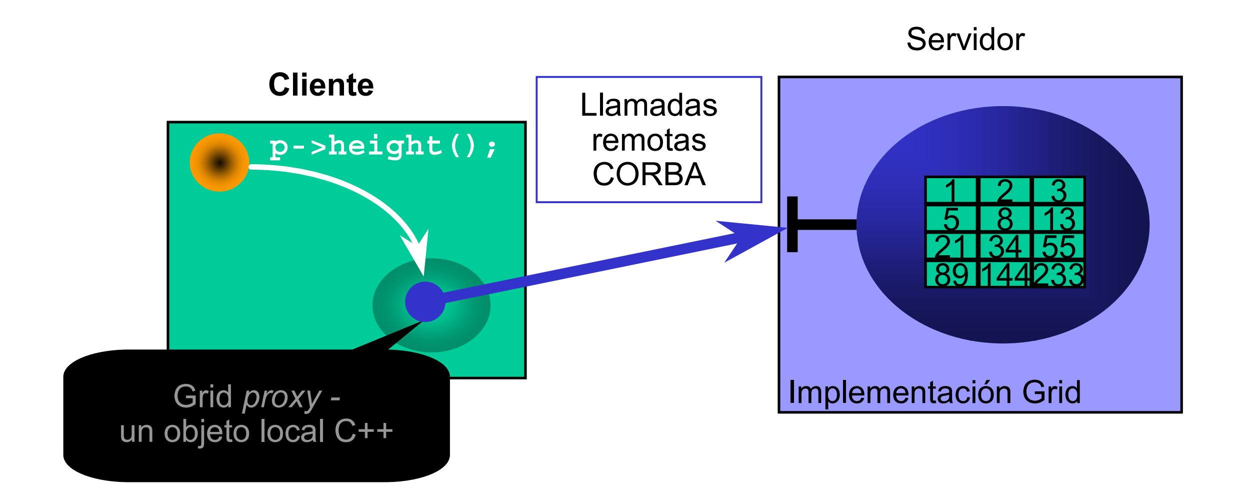
main () {
  Grid_var p;                  // como un puntero de C++

    // Se conecta aun objeto grid remoto
  p = Grid::_bind("algunObj:algunServ", algunHost);

    // Ahora se usa como cualquier objeto

  cout << "la altura es " << p->height() << endl;
  cout << "la anchura es "  << p->width() << endl;
  p->set(2,4,123);
  cout << "grid[2,4] es " << p->get(2,4) << endl;
};
```

¿Cómo trabaja?



Ahora: La parte del servidor ...

- *Escribir* una Clase
- *Heredar* la funcionalidad del ORB
- *Añadir* los datos y funciones miembros
- *Redefinir* las funciones IDL

```
class grid_i    : public virtual gridBOAImpl

  {
          short m_height, m_width;

  public:
          height(...);
          width(...);
          … set(...);
          … get(...);
  }
```

En el interior del servidor

Programa principal del servidor

"server mainline"

```
main() {
  // Crear algunos objetos
  // (Sólo uno en este caso)

  grid_i myGrid(100,100);

  try {

  //Dar el control al ORB
  CORBA::Orbix.impl_is_ready();

} catch { .... }
  cout << "Servidor finalizado"
    << endl;
}
```

Servidor

1	2	3
5	8	13
21	34	55
89	144	233

Objeto Grid

¿Qué es un 'server mainline'?

- Un servidor es el lugar donde los objetos viven
- El 'server mainline' es el código que:
  - Crea los objetos iniciales en el servidor
  - Da el control al ORB para recibir peticiones
- Los objetos se ponen a trabajar ellos mismos

Registrando el servidor

- CORBA tiene dos “repositorios”
- Repositorio de implementaciones (Implementation Repository)
  - Almacena los ejecutables
  - Es donde el componente de activación arranca
- Repositorio de interfaces (Interface Repository)
  - Tiene muchos usos
  - Principalmente relacionados con el comportamiento dinámico y con la comprobación de tipos

3.12 Genericidad

Es la propiedad que permite construir abstracciones modelo (clases genéricas) para otras clases. El modelo puede parametrizarse con otras clases, objetos y/o operaciones.

[Booch 94]

- Es un elemento del modelo de objetos discutido, algunos autores como Booch lo consideran ligado al lenguaje de programación que implementa el modelo de objetos.
- C++ y Eiffel soportan mecanismos de clases genéricas, habitualmente denominados clases parametrizadas o plantillas (templates)
- Java y C# no soportan este tipo de genericidad (utilizan genericidad basada en polimorfismo).
- En C y Pascal pueden construir estructuras de datos genéricas utilizando punteros genéricos (void y pointer), pero sin ninguna comprobación de tipos.
- B. Meyer *[Meyer 97]* ha apuntado que la herencia es un mecanismo más potente que la genericidad y que gran parte de los beneficios de la genericidad puede conseguirse mediante la herencia, pero no al revés.
- En la práctica, es útil usar un lenguaje que soporte tanto la herencia como clases parametrizadas
- Actualmente C++ soporta la nueva norma ANSI con la biblioteca de clases genéricas std

Ejemplo de genericidad en C++ utilizando plantillas (*templates*) (I)

[Stroustrup 2000, cap. 13]

Vector | T

UML

```
// TemplateVector.h
// Ejemplo de uso de plantillas
// Versión 2.0, 1 - Diciembre - 2002
// Autor: J.M. Cueva Lovelle

template <class T> class Vector
{
protected:
  T *dato;
  int tamagno;
public:
   Vector(int);
   ~Vector( ) { delete[ ] dato; }
   T& operator[ ] (int i) { return dato[i]; }
   int verTamagno(void){return tamagno;}
};

template <class T> Vector<T>::Vector(int n)
{
  dato = new T[n];
  tamagno = n;
};
```

Ejemplo de genericidad en C++ utilizando plantillas (*templates*) (II)

```
// PlantillaVector.cpp
// Ejemplo de uso de templates
// Versión 2.0, 1 - Diciembre - 2002
// Autor: J.M. Cueva Lovelle
// Compilado con gcc de GNU
// $g++ -o PlantillaVector.out PlantillaVector.cpp
// -----------------------------------

#include <iostream>
using namespace std;

#include "TemplateVector.h"

int main()
{
 Vector<int> x(10);    //Crea un vector con 10 enteros
 for (int i = 0; i < x.verTamagno(); ++i)
  {
   x[i] = i;           // Inicializa el  vector.
   cout<<x[i]<<endl;   // Escribe el vector
   }
return 0;
}
```

Ejemplo de genericidad en C++ utilizando plantillas (*templates*) (III) Versión en C++ Builder

```
// PlantillaVector.cpp
// Ejemplo de uso de templates
// Versión 2.0, 1 - Diciembre - 2002
// Autor: J.M. Cueva Lovelle
// Compilado con C++ Builder 4.0
// -----------------------------------
#pragma hdrstop
#include <condefs.h>
#include <iostream>
#include <conio>
using namespace std;

#include "TemplateVector.h"

#pragma argsused
int main(int argc, char* argv[])
{
 Vector<int> x(10);    //Crea un vector con 10 enteros
 for (int i = 0; i < x.verTamagno(); ++i)
  {
   x[i] = i;           // Inicializa el  vector.
   cout<<x[i]<<endl;   // Escribe el vector
   }
getch();
return 0;
}
```

Contenedores de la biblioteca estándar de C++ [Stroustrup 2000, cap. 16]

- La biblioteca estándar se define en el espacio de nombres `std`
- Un contenedor es un objeto que contiene otros objetos
- Contenedores de la biblioteca estándar
    - `<vector>` *array unidimensional de T*
    - `<list>` *lista doblemente enlazada de T*
    - `<deque>` *cola de doble extremo de T*
    - `<queue>` *cola de T*
    - `<stack>` *pila de T*
    - `<map>` *array asociativo de T*
    - `<set>` *conjunto de T*
    - `<bitset>` *array de booleanos*
- En general, se pueden añadir y eliminar objetos de un contenedor
- Los contenedores son por defecto seguros en tipos y homogéneos (todos los elementos del contenedor son del mismo tipo)
- Se puede proporcionar un contenedor heterogéneo como un contenedor homogéneo de punteros a una base común.

Contenedores de la biblioteca estándar de C++
Ejemplo de uso de <vector> en gcc (GNU)

```
// ContenedorVector.cpp
// Uso del contenedor <vector> de la biblioteca estándar
// Versión 2.0, 1 - Diciembre - 2002
// Autor: J.M. Cueva Lovelle
// Compilado con gcc de GNU
// -----------------------------------

#include <iostream>
#include <vector>    //Para usar el contenedor vector
using namespace std;

int main()
{
 vector<int> x(10);    //Crea un vector con 10 enteros
 for (unsigned i = 0; i < x.size(); ++i)
  {
   x[i] = i;           // Inicializa el  vector.
   cout<<x[i]<<endl;   // Escribe el vector
   }
return 0;
}
```

Contenedores de la biblioteca estándar de C++ Ejemplo de uso de <vector> en C++ Builder

```
// ContenedorVector.cpp
// Uso del contenedor <vector> de la biblioteca estándar
// Versión 2.0, 1 - Diciembre - 2002
// Autor: J.M. Cueva Lovelle
// Compilado con C++ Builder 4.0
// -----------------------------------

#pragma hdrstop
#include <condefs.h>
#include <conio>
#include <iostream>
#include <vector>    //Para usar el contenedor vector
using namespace std;

#pragma argsused
int main(int argc, char* argv[])
{
 vector<int> x(10);    //Crea un vector con 10 enteros
 for (unsigned i = 0; i < x.size(); ++i)
  {
   x[i] = i;           // Inicializa el  vector.
   cout<<x[i]<<endl;   // Escribe el vector
   }
 getch();
 return 0;
}
```

Contenedores de la biblioteca estándar de C++ Ejemplo de uso de <list> en C++ con gcc

```
// ContenedorLista.cpp
// Uso del contenedor <lista> de la biblioteca estándar
// Versión 2.0, 1 - Diciembre - 2002
// Autor: J.M. Cueva Lovelle
// Compilado con gcc de GNU
// $ g++ -o ContenedorLista.out ContenedorLista.cpp
// ------------------------------------
#include <string>
#include <iostream>
#include <list>
using namespace std;
ostream& operator<<(ostream& out, const list<string>& l)
 {
   copy(l.begin(), l.end(), ostream_iterator<string>(cout," "));
   return out;
 }
int main()
{
 list<string> animales;
 animales.push_front("Vaca"); //insertar en cabeza
 animales.insert(animales.begin(),"Perro"); //insertar en cabeza
 cout<<animales<<endl;
 animales.insert(animales.begin(),"Gato"); //insertar en cabeza
 animales.insert(animales.begin(),"Gallina"); //insertar en cabeza
 animales.push_back("Rata"); //insertar en cola
 animales.push_back("Elefante"); //insertar en cola
 animales.push_front("Rinoceronte"); //insertar en cabeza
 cout<<animales<<endl;
 cout<<"Lista desordenada: "<<animales<<endl;
 cout<<"Cabeza de lista: "<<animales.front()<<endl;
 cout<<"Cola de lista: "<<animales.back()<<endl;

 //Eliminar un elemento de la lista
 animales.remove("Pingüino");
 cout<<"Lista desordenada: "<<animales<<endl;

 animales.sort();
 cout<<"Lista ordenada: "<<animales<<endl;

 // Borrar media lista
 int mitad = animales.size() >> 1; // rota bits para dividir por 2
   for(int i = 0; i < mitad; ++i) {
     animales.erase(animales.begin());
   }
 cout<<"Mitad de lista: "<<animales<<endl;
 // Borrar la lista
 animales.clear();
 cout<<"Lista vacia: "<<animales<<endl;
 cout<<"Tamaño: "<<animales.size()<<endl;
 return 0;
}
```

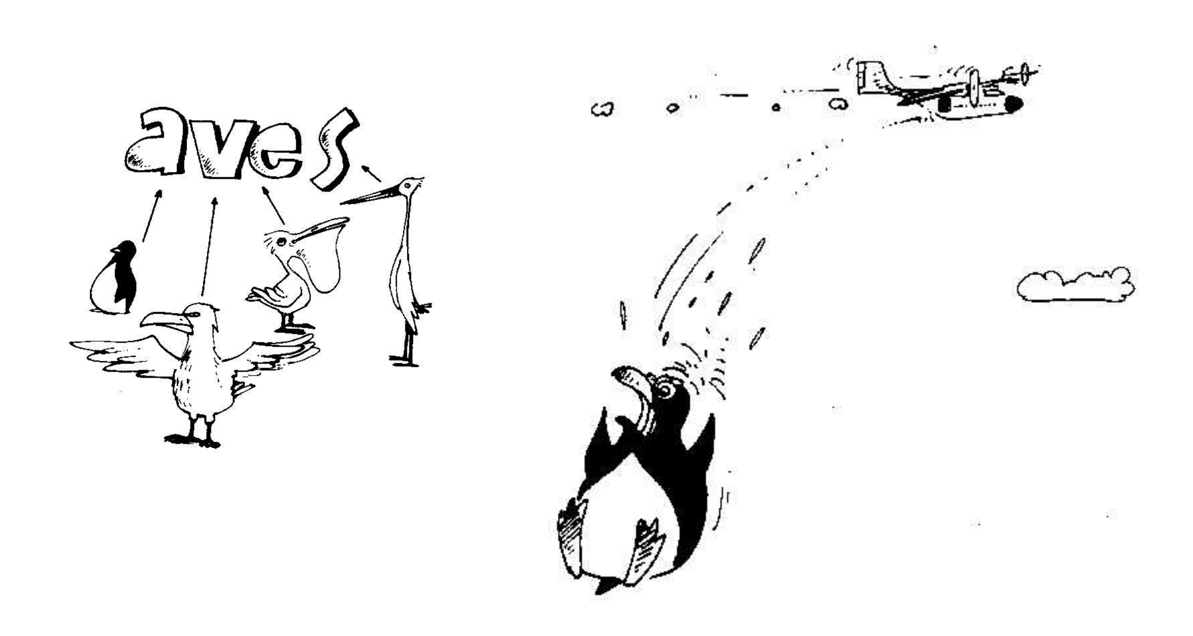
Contenedores de la biblioteca estándar de C++
Ejemplo de uso de <list> en C++ Builder

```
// ContenedorLista.cpp
// Uso del contenedor <lista> de la biblioteca estándar
// Versión 2.0, 1 - Diciembre - 2002
// Autor: J.M. Cueva Lovelle
// Compilado con C++ Builder 4.0
#pragma hdrstop
#include <condefs.h>
#include <conio>
#include <string>
#include <iostream>
#include <list>
using namespace std;
ostream& operator<<(ostream& out, const list<string>& l)
 {copy(l.begin(), l.end(), ostream_iterator<string>(cout," "));
  return out;}
#pragma argsused
int main(int argc, char* argv[])
{
 list<string> animales;
 animales.push_front("Vaca"); //insertar en cabeza
 animales.insert(animales.begin(),"Perro"); //insertar en cabeza
 cout<<animales<<endl;
 animales.insert(animales.begin(),"Gato"); //insertar en cabeza
 animales.insert(animales.begin(),"Gallina"); //insertar en cabeza
 animales.push_back("Rata"); //insertar en cola
 animales.push_back("Elefante"); //insertar en cola
 animales.push_front("Rinoceronte"); //insertar en cabeza
 cout<<animales<<endl;
 // Buscando en la lista e insertando delante
 animales.insert(find(animales.begin(),animales.end(),
                  "Rinoceronte"),"Pingüino");
 cout<<"Lista desordenada: "<<animales<<endl;
 cout<<"Cabeza de lista: "<<animales.front()<<endl;
 cout<<"Cola de lista: "<<animales.back()<<endl;
 //Eliminar un elemento de la lista
 animales.remove("Pingüino");
 cout<<"Lista desordenada: "<<animales<<endl;
 animales.sort();
 cout<<"Lista ordenada: "<<animales<<endl;
 // Borrar media lista
 int mitad = animales.size() >> 1; // se rotan bits para dividir por 2
   for(int i = 0; i < mitad; ++i) {
     animales.erase(animales.begin());}
 cout<<"Mitad de lista: "<<animales<<endl;
 // Borrar la lista
 animales.clear();
 cout<<"Lista vacía: "<<animales<<endl;
 cout<<"Tamaño: "<<animales.size()<<endl;
 getch();
 return 0;
}
```

3.13 Manejo de excepciones

Se deben tener mecanismos para recuperar el sistema de situaciones anormales no esperadas [Meyer 97]

- Los lenguajes orientados a objetos establecen mecanismos para que en el caso de situaciones anómalas se disparen excepciones que el programador puede atrapar en distintas partes del código
- Ejemplos:
  - C++, Eiffel, Java, C#, Delphi, VBASIC.NET, ...
- Una excepción es un objeto de alguna clase que representa un suceso excepcional *[Stroustrup 2000, cap. 14]*
- C++ utiliza las palabras reservadas `try`, `catch` y `throw` para el tratamiento de excepciones.
- La filosofía del manejo de excepciones es dispara y captura (*throw and catch*)



Ejemplo de manejo de excepciones en C++ (I)

```
// Excepciones.cpp
// Ejemplo de uso de excepciones
// Versión 1.0, 28 - Noviembre - 2002
// Autor: J.M. Cueva Lovelle
// Compilado con gcc de GNU (www.gnu.org)
//$ g++ -o Excepciones.out Excepciones.cpp
//----------------------------------------------------------
#include "Aves.h"
int main()
{
 Pinguino pigui("Pigui");
 try
 {
  pigui.volar();
 } catch(exception& e)
    {
    cout<<"Atrapada una excepción: "<<e.what()<<endl;
    pigui.abrirParacaidas();
    }
 return 0;
} // Fin del main
```

EJECUCIÓN

```
Ha nacido el pingüino Pigui
Atrapada una excepción: Pigui dice aah ...me caigo
Paracaidas abierto
Ha muerto el pingüino Pigui
Se ha matado a una ave
```

Ejemplo de manejo de excepciones en C++ (II)

```
// Aves.h
// Ejemplo de uso de excepciones
// Clases Ave, Pinguino
// Versión 1.0, 28 - Noviembre - 2002
// Autor: J.M. Cueva Lovelle
#ifndef AVES_HPP
#define AVES_HPP
#include <iostream>  //para entrada/salida
#include <string>    //para manejo de string
#include <stdexcept> //para crear objetos excepción
using namespace std; //para usar la biblioteca estándar
class Ave {     //clase abstracta Ave
public:
 virtual void volar(void)=0;  //método virtual puro
//Para asegurar la destrucción de objetos de clases derivadas
 virtual ~Ave(){cout<<"Se ha matado a una ave"<<endl;};
};
class Pinguino: public Ave {
 string nombre;
public:
 Pinguino(string unNombre){ nombre=unNombre;
             cout<<"Ha nacido el pingüino "<<nombre<<endl;};
 void volar(void)
     {throw runtime_error (nombre+" dice aah ...me caigo");};
 void abrirParacaidas (void)
          {cout<<"Paracaidas abierto"<<endl;};
~Pinguino(void)
          {cout<<"Ha muerto el pingüino"<<nombre<<endl;};
};
#endif
```

Ejemplo de manejo de excepciones en C++ (III)

```
// Excepciones.cpp
// Ejemplo de uso de excepciones
// Versión 1.0, 28 - Noviembre - 2002
// Autor: J.M. Cueva Lovelle
// Compilado con C++ Builder 4.0
//-----------------------------------------------------------
#pragma hdrstop
#include <condefs.h>

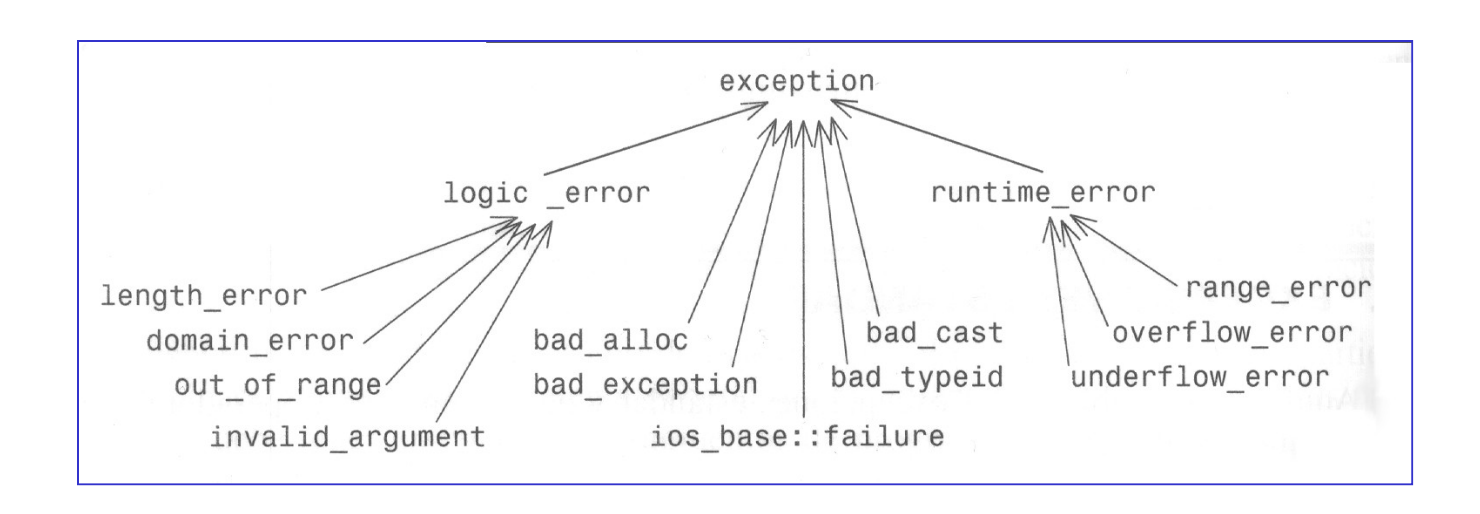
#include <conio> //para uso de getch()

#include "Aves.h"
#pragma argsused
int main(int argc, char* argv[])
{
 Pinguino pigui("Pigui");
 try
 {
  pigui.volar();
 } catch(exception& e)
    {
    cout<<"Atrapada una excepción: "<<e.what()<<endl;
    pigui.abrirParacaidas();
    }
 getch();
 return 0;
} // Fin del main
```

Excepciones estándar en C++

[Stroustrup 2000, cap. 14]

- Las excepciones estándar derivan de la clase `exception`, que se encuentra en `<stdexcept>`
- No todas las excepciones derivan directamente de la clase `exception`
- Se pueden crear clases excepción derivadas de la clase `exception` o sus derivadas



Interfaces de la clase exception y sus derivadas en C++

```
class exception {
  public:
    exception () throw();
    exception (const exception&) throw();
    exception& operator= (const exception&) throw();
    virtual ~exception () throw();
    virtual const char* what () const throw(); };
 class logic_error : public exception {
  public:
    explicit logic_error (const string& what_arg); };
 class domain_error : public logic_error {
  public:
    explicit domain_error (const string& what_arg);};
 class invalid_argument : public logic_error {
  public:
    explicit invalid_argument (const string& what_arg);};
 class length_error : public logic_error {
  public:
    explicit length_error (const string& what_arg);};
 class out_of_range : public logic_error {
  public:
    explicit out_of_range (const string& what_arg);};
 class runtime_error : public exception {
  public:
    explicit runtime_error (const string& what_arg); };
 class range_error : public runtime_error {
  public:
    explicit range_error (const string& what_arg);};
 class overflow_error : public runtime_error {
  public:
    explicit overflow_error (const string& what_arg);};
 class underflow_error : public runtime_error {
  public:
    explicit underflow_error (const string& what_arg);};
```

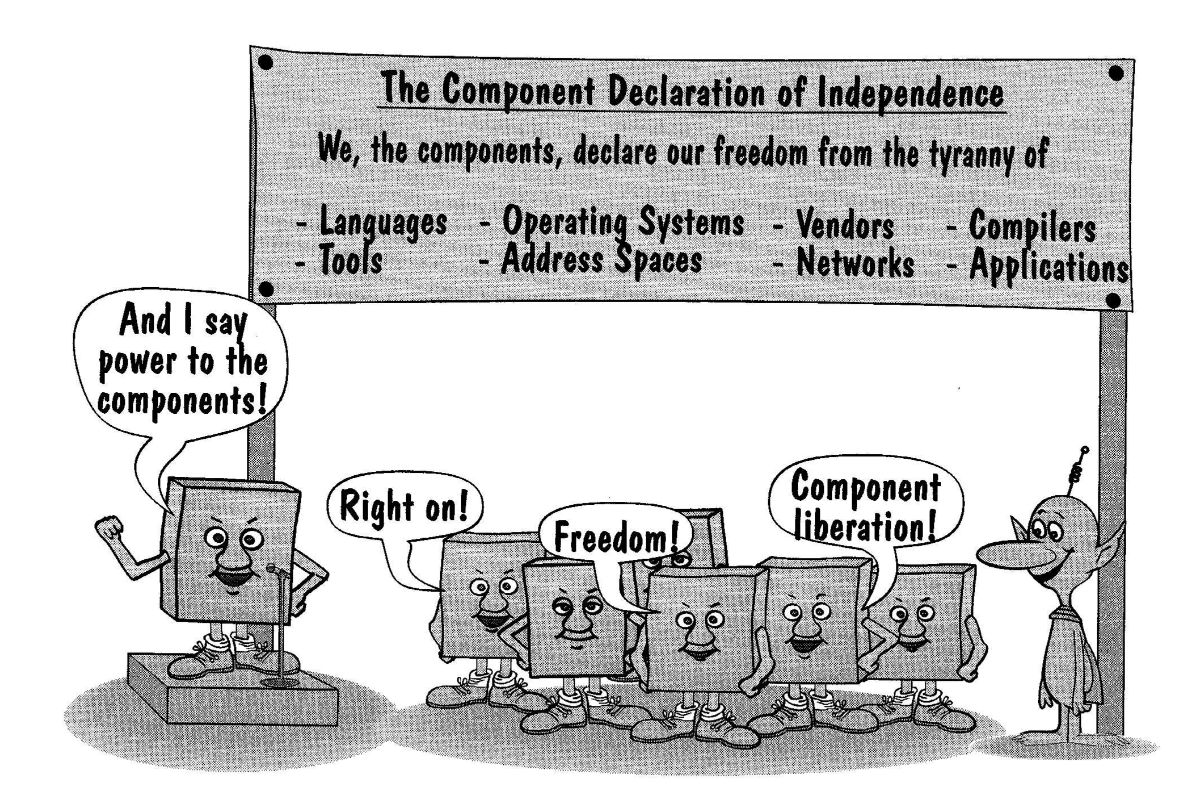
Elementos del modelo de objetos

- **Elementos fundamentales**
  - ✓ Abstracción
  - ✓ Encapsulación y ocultación de la información
  - ✓ Modularidad
  - ✓ Jerarquía (herencia, polimorfismo y agregación)
- **Elementos secundarios**
  - ✓ Tipos (control de tipos)
  - ✓ Concurrencia
  - ✓ Persistencia
  - ✓ Distribución
  - ✓ Sobrecarga
  - ✓ Genericidad
  - ✓ Manejo de excepciones
- **Elementos relacionados**
  - – Componentes
  - – Patrones de diseño

3.14 Componentes

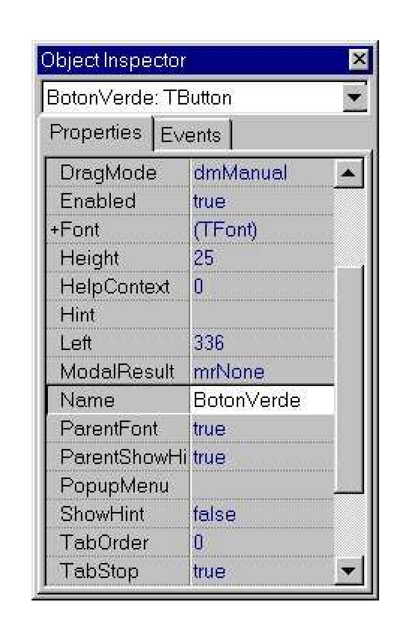
Un componente es un elemento de software que por una parte debe ser suficientemente pequeño para crearse y mantenerse y por otra suficientemente grande para poder utilizarse, además debe tener interfaces estándar para que sea interoperable [Orfali 96]

- Propiedades de los componentes
  - Son entidades que se pueden comercializar separadas
  - No son aplicaciones completas
  - Pueden combinarse en formas impredecibles
  - Tienen un interfaz perfectamente especificado
  - Son objetos interoperables. Se pueden invocar de forma independiente de los sistemas software a través del espacio de direcciones, redes, lenguajes, sistemas operativos y herramientas
  - Son una extensión de los objetos. Es decir deben soportar encapsulación, herencia, polimorfismo,...



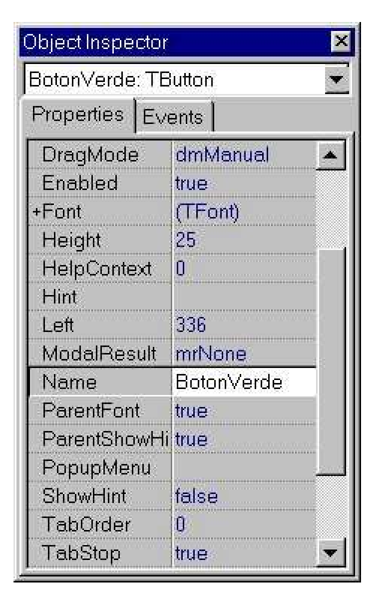
Componentes

- Un componente es una entidad funcional que está completamente caracterizada por sus entradas y salidas
- Un componente puede usarse y comprobarse como una unidad, independiente del contexto en el que se esté usando eventualmente
- La implementación interna de un componente está oculta al usuario
- Un componente está formado por
  - Propiedades
  - Métodos
  - Eventos



Propiedades

- Una propiedad es un valor, o estado, asociado al componente
- Las propiedades no son solamente variables cuyo valor se puede leer o escribir directamente, como se haría en una clase C++
- Las propiedades tienen normalmente asociada una acción que se ejecuta con la modificación de la propiedad
- Las propiedades tienen un valor antes de que se use el componente
- El valor de una propiedad puede cambiarse en tiempo de ejecución
- Los resultados de modificar una propiedad son inmediatos al desencadenarse la acción ligada a la propiedad

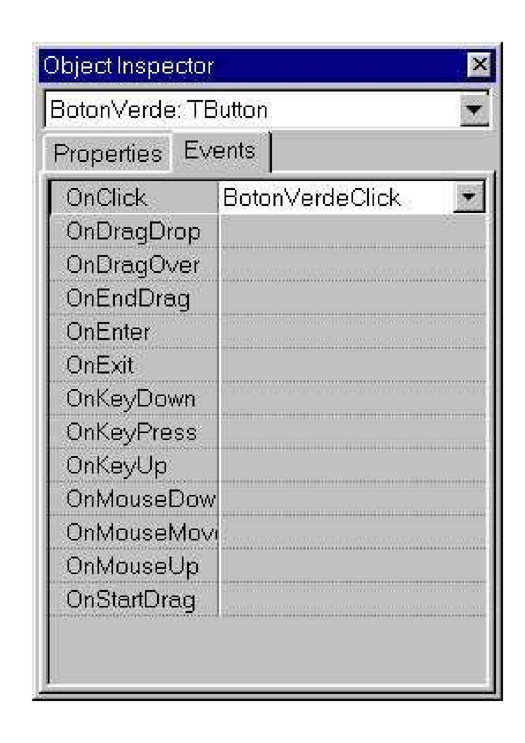


Métodos

- Los métodos de un componente funcionan como lo haría cualquier método de una clase de C++
- Cada método proporciona un comportamiento de un componente, ya que obligan al componente a realizar una acción

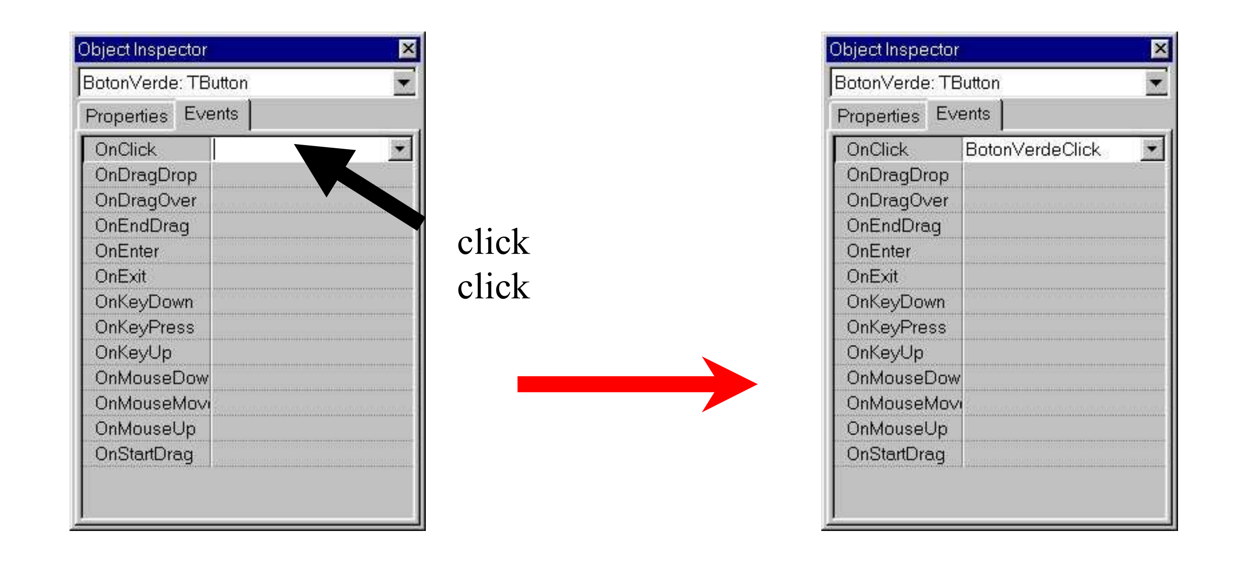
Eventos o sucesos

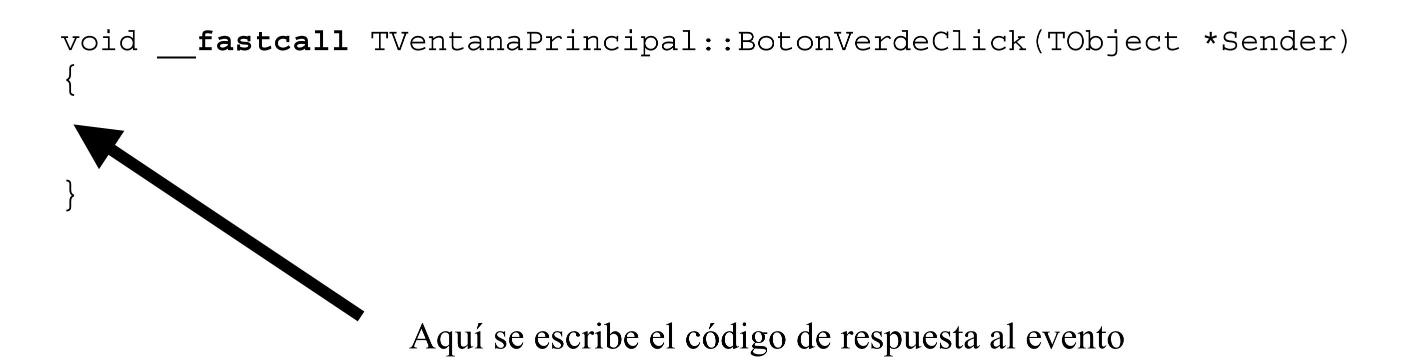
- Un evento es alguna acción producida por el usuario o por actividades internas del sistema
- Ejemplos:
  - pulsar el ratón
  - llamar a un método
  - modificar el valor de una propiedad



Manejo de eventos

- Es el código el que responde al evento cuando éste ocurre
- El programador escribirá el código adecuado para responder a cada evento





Tipos de componentes

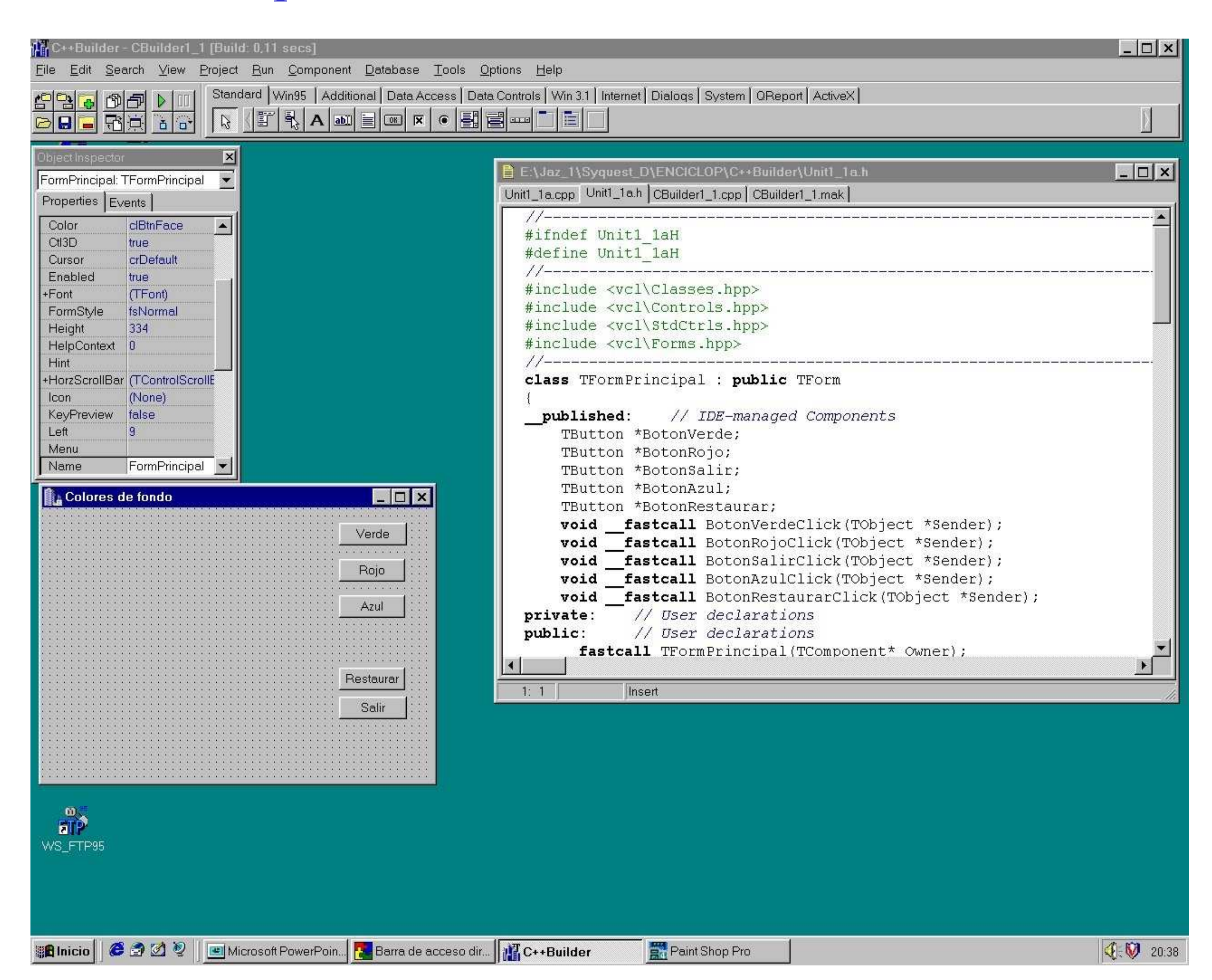
[Szyperski 1997] [Chauvet 1997][Charte 2001]

- **ActiveX**
  - Es la arquitectura de componentes de Microsoft para los sistemas operativos de la familia Windows.
  - Los componentes ActiveX son independientes del lenguaje de programación que los utiliza pero deben utilizarse dentro de la familia de sistemas operativos Windows.
  - Nace como una evolución de **OLE** (*Object Linking and Embedding)* desarrollado por Microsoft para "enlazar e incrustar objetos" entre sus aplicaciones ofimáticas (Word, Excel, Power Point, etc.) dentro del sistema operativo Windows.
  - En un principio se desarrollaron componentes de *16 bits* denominados **VBX**, donde VB es la abreviatura de *Visual Basic* primer producto de Microsoft que los utilizó. Aunque son compatibles con cualquier lenguaje de programación (Delphi, Visual C++, etc.)
  - Posteriormente se desarrollaron los componentes de *32 bits* denominados en un principio **OCX** y que incorporaban el modelo de objetos de Microsoft denominado **COM** *(Component Object Model).*
  - Microsoft revisó la especificación de los OCX para competir con otros componentes que se viajaban a través de Internet y presentó los componentes de *32 bits* denominados **ActiveX**, que tienen auto-registro en el servidor donde se utilizan (por ejemplo si se descargan por Internet).
  - Los componentes ActiveX son objetos COM soportados por un servidor especial.
- **Componentes en .NET Framework**
  - Es un nuevo tipo de componentes desarrolado por Microsoft para la máquina virtual CLR
- **VCL** *(Visual Component Library)* y **CLX** *(Cross platform Library)*
  - VCL Componentes desarrollados por Borland para trabajar con sus compiladores (Delphi, C++Builder) para la familia de sistemas operativos Windows.
  - Las herramientas de Borland permiten transformar los componentes VCL en ActiveX o utilizar directamente los ActiveX en Windows
  - Los componentes CLX trabajan en el Sistema Operativo Linux y las ultimas versiones de los compiladores de Borland para Windows. Inicialmente salieron con Kylix (compilador equivalente a Delphi para Linux) y con Delphi 6 para Windows.
- **Java Beans**
  - Son los componentes de Sun para Java que se añadieron a Java en Octubre de 1996
  - Sólo pueden utilizarse en Java aunque bajo cualquier sistema operativo.
  - Los componentes Java Beans se caracterizan por
    - **Propiedades**
    - **Eventos**
    - **Métodos**
    - **Introspección** *(Introspection)*
      - *Un bean puede ser inspeccionado por cualquier herramienta de ensamblaje mostrándose las propiedades, eventos y métodos que soporta*
    - **Personalización** *(Customization)*
      - *Un bean puede personalizarse mediante una herramienta de ensamblaje estableciendo nuevas propiedades*
    - **Persistencia**
      - **Los beans se pueden serializar (crear un stream de bytes) y deserializar (devolverlo a su estado original). Esta característica es interesante para cuando los bean viajan por Internet**

C++ Builder

Producto de Borland caracterizado por
•Compilador de C++ ANSI
•Desarrollo rápido de aplicaciones (RAD, *Rapid Application Development*) para entornos Windows 95/NT
•Entorno de desarrollo basado en componentes **ActiveX** y componentes **VCL** *(Visual Component Library)*
•Permite crear componentes
•Tiene componentes para el manejo de bases de datos relacionales
•Incluye extensiones a C++ para facilitar el desarrollo

Aspecto del entorno C++ Builder



Creación de una aplicación en C++ Builder

Se construye una aplicación que al pulsar un botón se cambia el color de fondo de la ventana al color seleccionado por el botón. El botón restaurar vuelve a colocar el color de fondo inicial de la ventana. El botón salir finaliza la aplicación.



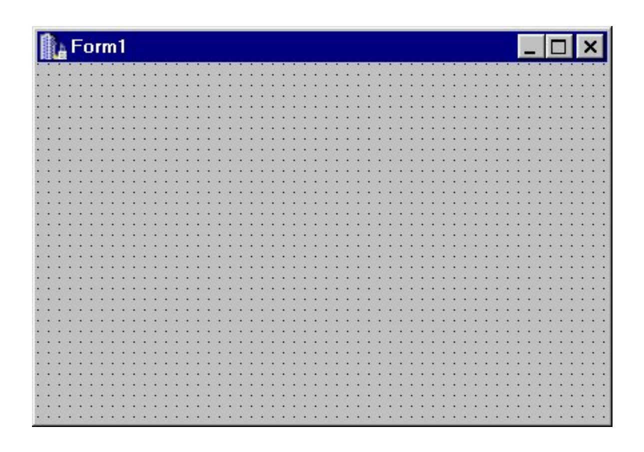
Se pulsa el botón Verde y se coloca el fondo verde



Creación de una Ficha (Form)

*Las **fichas** (form) son las ventanas del entorno Windows en las que se alojarán los distintos controles encargados de interactuar con el usuario*

Por defecto *C++Builder* define a la primera ficha como ***Form1*,** que aparece de forma visual en la pantalla tal como se muestra a continuación:



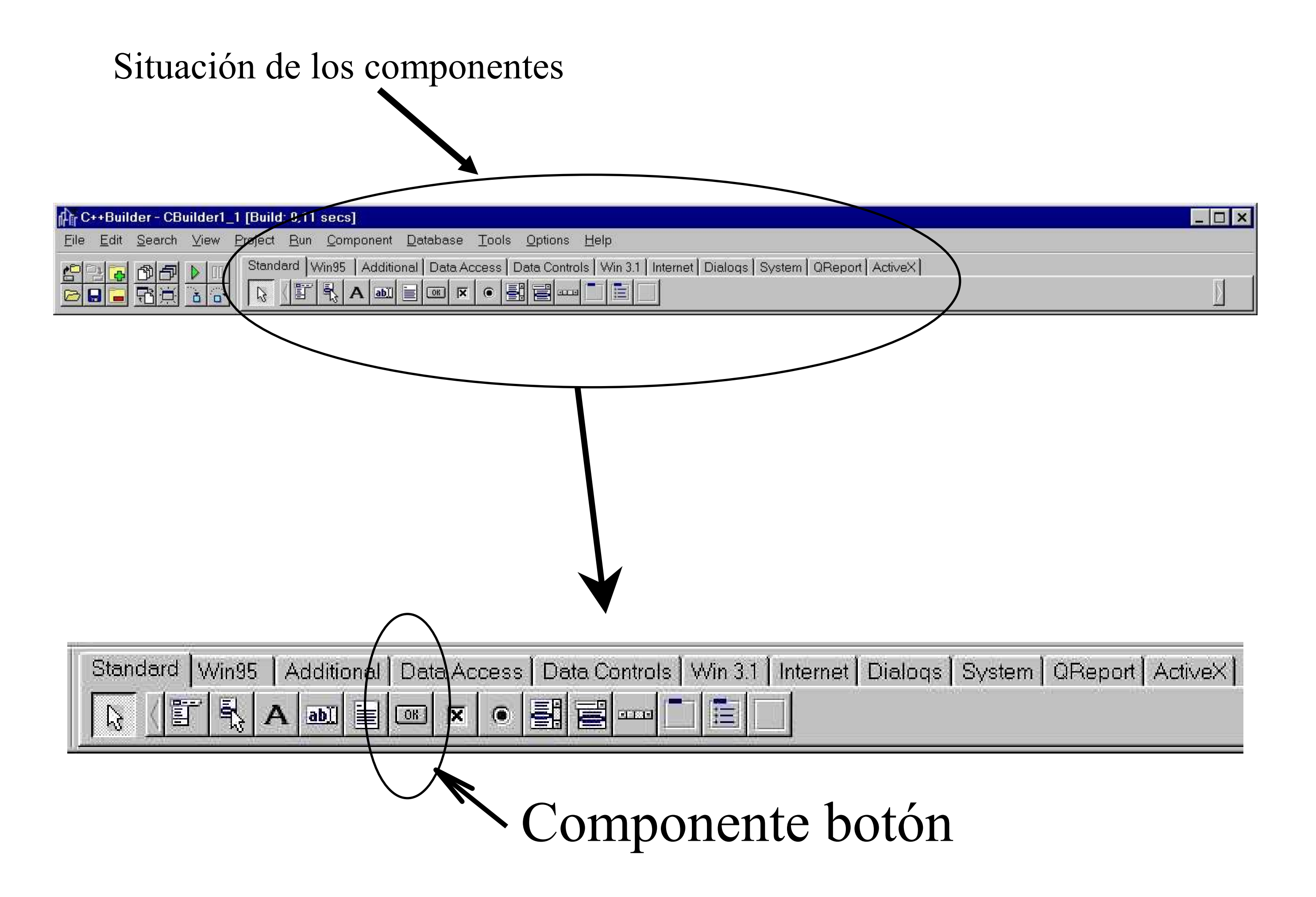
Form1 es un puntero a un objeto de la clase ***TForm1***, y el código se genera automáticamente de la siguiente forma

```
...
TForm1 *Form1;
```

La clase **TForm1** es una clase que hereda de **TForm**

```
class TForm1 : public TForm
{
...
}
```

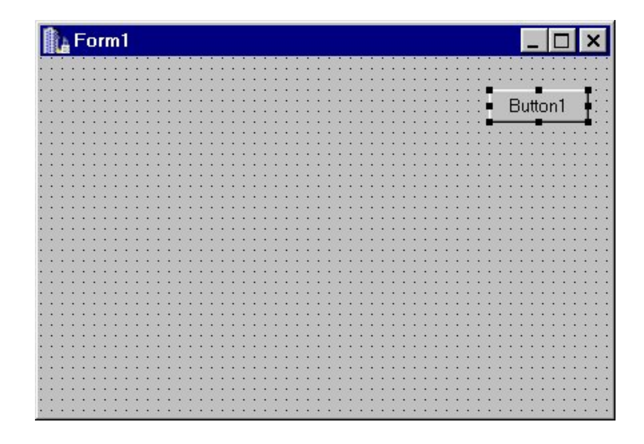
Componentes en C++ Builder



Los componentes se arrastran y se insertan en las fichas (Form), tanto si su función es visual o si no lo es.

Agregación de componentes

Para agregar un botón se selecciona un botón del panel y se arrastra a la ficha



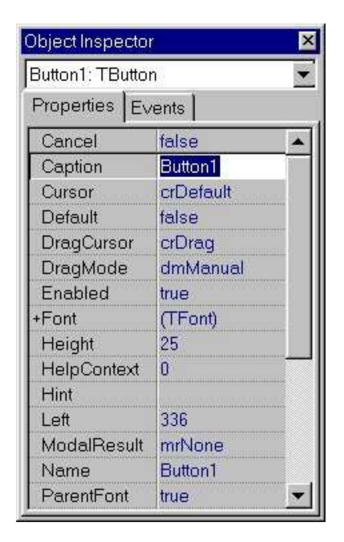
El código que se genera corresponde a una agregación en la clase **TForm1** de un puntero a un objeto de la clase **TButton**

```
class TForm1 : public TForm
{
__published:     // IDE-managed Components
        TButton *Button1;
private: // User declarations
public:  // User declarations
        __fastcall TForm1(TComponent* Owner);
};
```

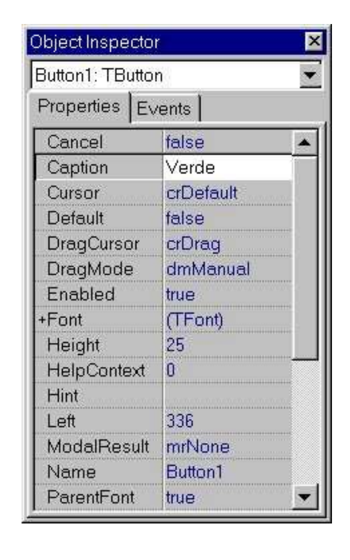
El inspector de objetos (*Object Inspector*)

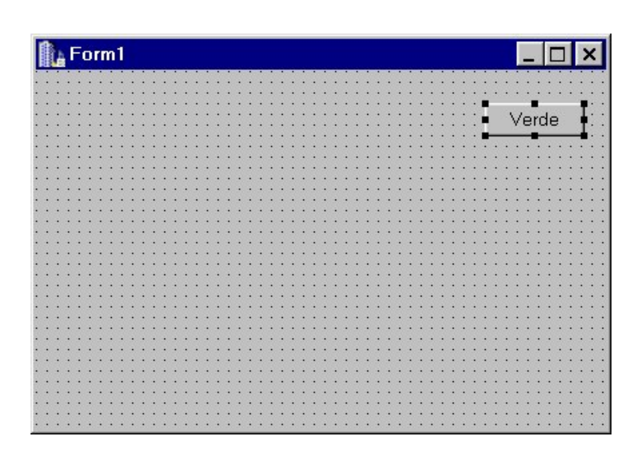
El **inspector de objetos** es un elemento del entorno de *C++ Builder* que *tiene como finalidad facilitar la edición de las propiedades y eventos del objeto que se seleccione.*

Así si se selecciona el objeto **Button1**, se observa una lista de las propiedades de dicho objeto, así como los valores iniciales de estas propiedades.



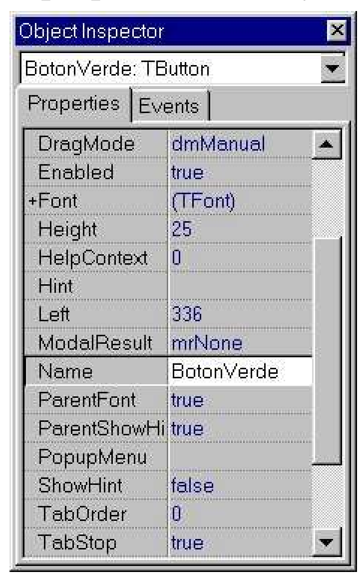
Las propiedades se pueden modificar directamente en el inspector de objetos. Por ejemplo la propiedad ***Caption*** muestra el texto del botón, por defecto su valor es **Button1**, se cambiará a **Verde**, tanto en el inspector de objetos como en la ficha.





El inspector de objetos (*Object Inspector)*

También se puede modificar en el inspector de objetos la propiedad ***Name*** que es el identificador del objeto.
Así si se desea que el objeto **Botton1** se denomine **BotonVerde** se modifica la propiedad ***Name*** y automáticamente se modifica en el código fuente.




```
class TForm1 : public TForm
{
__published:
 TButton *BotonVerde;
private:
public:
  __fastcall TForm1(TComponent* Owner);
};
```

También se puede modificar la propiedad ***Name*** del objeto **Form1** a **VentanaPrincipal,** modificándose tanto el identificador del objeto **Form1** como el de la clase **TForm1.** Modificándose el código fuente:

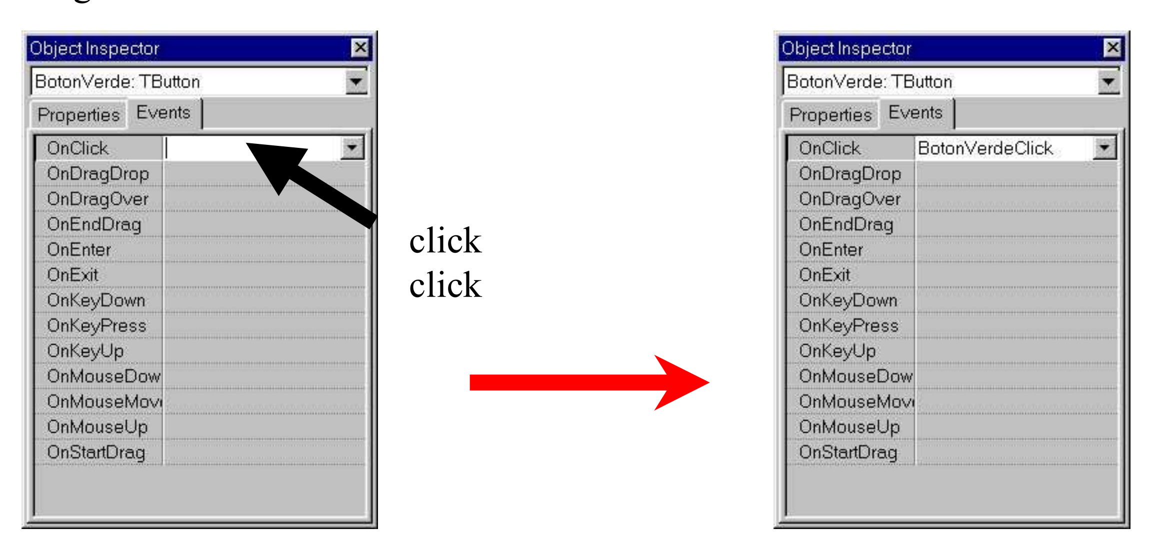
Object Inspector
VentanaPrincipal: TVentanaPrincipal
Properties | Events

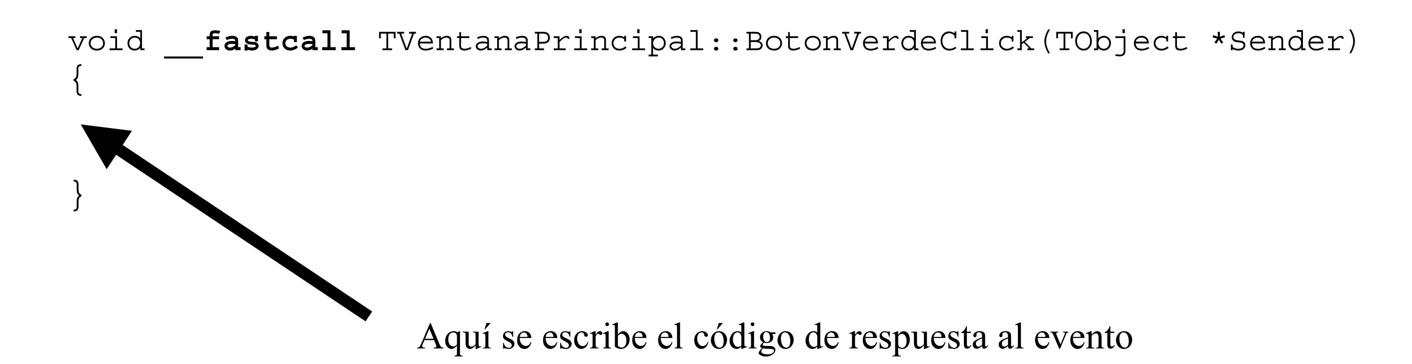
+Font	(TFont)
FormStyle	fsNormal
Height	300
HelpContext	0
Hint	
+HorzScrollBar	(TControlScrollBar)
Icon	(None)
KeyPreview	false
Left	687
Menu	
Name	VentanaPrincipal
ObjectMenuIte	
PixelsPerInch	120
PopupMenu	
Position	poDesigned

```
class TVentanaPrincipal : public TForm
{
__published:
    TButton *BotonVerde;
private:
public:
  __fastcall TVentanaPrincipal(TComponent*
Owner);
};
//---------------------------------------
extern TVentanaPrincipal *VentanaPrincipal;
//---------------------------------------
```

El inspector de objetos (*Object Inspector*) *Eventos*

•Un **evento** es una señal, externa o interna a la aplicación, que produce la ejecución de un bloque de código que se escribe como un método.
•Cada componente tiene definidas propiedades y eventos. Los eventos están definidos, pero el bloque de respuesta de código no está escrito.
•A continuación se muestran los eventos del objeto **BotonVerde**, que se obtienen levantando la pestaña ***Events*** del inspector de objetos.
•Para escribir el código de respuesta al evento de hacer click sobre el botón (***OnClick***), se debe hacer doble click con el ratón sobre el campo de edición de *OnClick,* y se nos crea automáticamente el método **`BotonVerdeClick`**, y el cursor se posiciona en el editor de código fuente para escribir el bloque de código asociado al evento:





Escribiendo el código de respuesta a un evento

• Se desea que cuando se pulse el botón Verde se cambie la propiedad ***Color*** de ***TVentanaPrincipal*** a verde usando el tipo enumerado ***clGreen***. Se escribe entonces la línea de código:

```
void __fastcall TVentanaPrincipal::BotonVerdeClick(TObject *Sender)
{
  Color=clGreen;
}
```

•Ahora ya se puede compilar el programa y ejecutar para hacer una prueba

•De la misma forma se añaden el resto de los componentes botón y escribe el código de respuesta para que cambien la propiedad ***Color*** de ***TVentanaPrincipal.*** Para cerrar la ventana se utiliza el método ***Close()***

```
void __fastcall TVentanaPrincipal::BotonRojoClick(TObject *Sender)
{
 Color=clRed;
}
//---------------------------------------------------------------------------
void __fastcall TVentanaPrincipal::BotonAzulClick(TObject *Sender)
{
 Color=clBlue;
}
//---------------------------------------------------------------------------
void __fastcall TVentanaPrincipal::BotonRestaurarClick(TObject *Sender)
{
 Color=clBtnFace;
}
//---------------------------------------------------------------------------
void __fastcall TVentanaPrincipal::BotonSalirClick(TObject *Sender)
{
 Close();
}
```

Código fuente de la cabecera del módulo *Unit1_1a.h*

```
//---------------------------------------------------------------------------
#ifndef Unit1_1aH
#define Unit1_1aH
//---------------------------------------------------------------------------
#include <vcl\Classes.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
//----------------------------------------------------------------------------
class TVentanaPrincipal : public TForm
{
__published:        // IDE-managed Components
          TButton *BotonVerde;
          TButton *BotonRojo;
          TButton *BotonSalir;
          TButton *BotonAzul;
          TButton *BotonRestaurar;
          void __fastcall BotonVerdeClick(TObject *Sender);
          void __fastcall BotonRojoClick(TObject *Sender);
          void __fastcall BotonSalirClick(TObject *Sender);
          void __fastcall BotonAzulClick(TObject *Sender);
          void __fastcall BotonRestaurarClick(TObject *Sender);
private:  // User declarations
public:             // User declarations
          __fastcall TVentanaPrincipal(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern TVentanaPrincipal *VentanaPrincipal;
//---------------------------------------------------------------------------
#endif
```

Código fuente del cuerpo del módulo *Unit1_1a.cpp*

```
//---------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include "Unit1_1a.h"
//---------------------------------------------------------------------
#pragma resource "*.dfm"
TVentanaPrincipal *VentanaPrincipal;
//---------------------------------------------------------------------
__fastcall TVentanaPrincipal::TVentanaPrincipal(TComponent* Owner)
          : TForm(Owner)
{
}
//---------------------------------------------------------------------
void __fastcall TVentanaPrincipal::BotonVerdeClick(TObject *Sender)
{
 Color=clGreen;
}
//--------------------------------------------------------------------
void __fastcall TVentanaPrincipal::BotonRojoClick(TObject *Sender)
{
 Color=clRed;
}
//-------------------------------------------------------------------
void __fastcall TVentanaPrincipal::BotonSalirClick(TObject *Sender)
{
 Close();
}
//-------------------------------------------------------------------
void __fastcall TVentanaPrincipal::BotonAzulClick(TObject *Sender)
{
 Color=clBlue;
}
//-------------------------------------------------------------------
void __fastcall TVentanaPrincipal::BotonRestaurarClick(TObject *Sender)
{
 Color=clBtnFace;
}
//----------------------------------------------------------------------
```

Código fuente del programa principal *CBuilder1_1.cpp*

```
//---------------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop
//---------------------------------------------------------------------------
USEFORM("Unit1_1a.cpp", VentanaPrincipal);
USERES("CBuilder1_1.res");
//---------------------------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
 try
  {
   Application->Initialize();
   Application->CreateForm(__classid(TVentanaPrincipal), &VentanaPrincipal);
   Application->Run();
  }
 catch (Exception &exception)
  {
   Application->ShowException(&exception);
  }
 return 0;
}
//---------------------------------------------------------------------------
```

Código fuente del fichero de compilación del proyecto *CBuilder1_1.mak*

```
# ---------------------------------------------------------------------------
VERSION = BCB.01
# ---------------------------------------------------------------------------
!ifndef BCB
BCB = $(MAKEDIR)\..
!endif
# ---------------------------------------------------------------------------
PROJECT = CBuilder1_1.exe
OBJFILES = CBuilder1_1.obj Unit1_1a.obj
RESFILES = CBuilder1_1.res
RESDEPEN = $(RESFILES) Unit1_1a.dfm
LIBFILES =
DEFFILE =
# ---------------------------------------------------------------------------
CFLAG1 = -Od -Hc -w -k -r- -y -v -vi- -c -a4 -b- -w-par -w-inl -Vx -Ve -x
CFLAG2 = -Ie:\jaz_1\syquest_d\enciclop;$(BCB)\include;$(BCB)\include\vcl \
   -H=$(BCB)\lib\vcld.csm
PFLAGS = -AWinTypes=Windows;WinProcs=Windows;DbiTypes=BDE;DbiProcs=BDE;DbiErrs=BDE \
   -Ue:\jaz_1\syquest_d\enciclop;$(BCB)\lib\obj;$(BCB)\lib \
   -Ie:\jaz_1\syquest_d\enciclop;$(BCB)\include;$(BCB)\include\vcl -v -$Y -$W \
   -$O- -JPHNV -M
RFLAGS = -ie:\jaz_1\syquest_d\enciclop;$(BCB)\include;$(BCB)\include\vcl
LFLAGS = -Le:\jaz_1\syquest_d\enciclop;$(BCB)\lib\obj;$(BCB)\lib -aa -Tpe -x \
   -v -V4.0
IFLAGS =
LINKER = ilink32
# ---------------------------------------------------------------------------
ALLOBJ = c0w32.obj $(OBJFILES)
ALLRES = $(RESFILES)
ALLLIB = $(LIBFILES) vcl.lib import32.lib cp32mt.lib
# ---------------------------------------------------------------------------
.autodepend

$(PROJECT): $(OBJFILES) $(RESDEPEN) $(DEFFILE)
    $(BCB)\BIN\$(LINKER) @&&!
    $(LFLAGS) +
    $(ALLOBJ), +
    $(PROJECT),, +
    $(ALLLIB), +
    $(DEFFILE), +
    $(ALLRES)
!

.pas.hpp:
    $(BCB)\BIN\dcc32 $(PFLAGS) { $** }

.pas.obj:
    $(BCB)\BIN\dcc32 $(PFLAGS) { $** }

.cpp.obj:
    $(BCB)\BIN\bcc32 $(CFLAG1) $(CFLAG2) -o$* $*

.c.obj:
    $(BCB)\BIN\bcc32 $(CFLAG1) $(CFLAG2) -o$* $**

.rc.res:
    $(BCB)\BIN\brcc32 $(RFLAGS) $<
#----------------------------------------------------------------------------
```

Formato texto del archivo de definición de componentes *Unit1_1a.dfm*

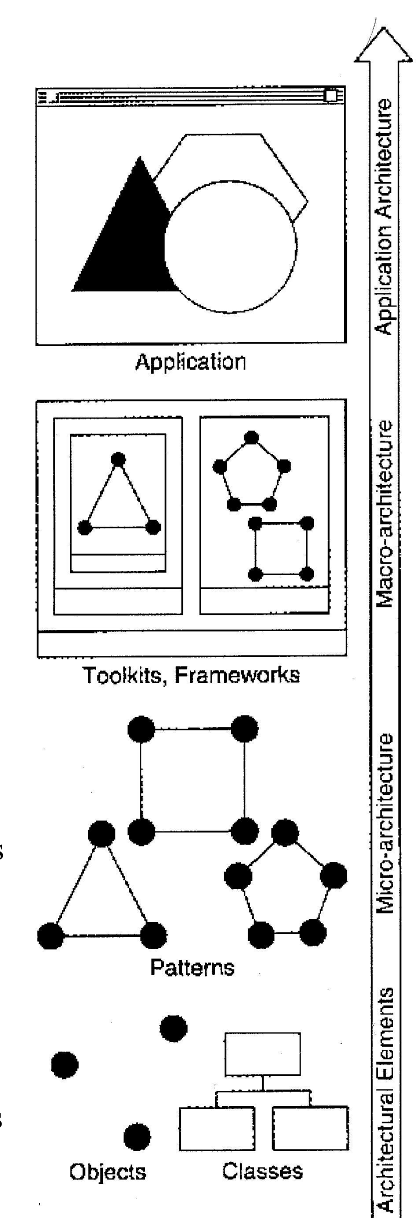
```
object VentanaPrincipal: TVentanaPrincipal
  Left = 708
  Top = 323
  Width = 440
  Height = 334
  Caption = 'Colores de fondo'
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -13
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  PixelsPerInch = 120
  TextHeight = 16
  object BotonVerde: TButton
    Left = 328
    Top = 16
    Width = 75
    Height = 25
    Caption = 'Verde'
    TabOrder = 0
    OnClick = BotonVerdeClick
  end
  object BotonRojo: TButton
    Left = 328
    Top = 56
    Width = 73
    Height = 25
    Caption = 'Rojo'
    TabOrder = 1
    OnClick = BotonRojoClick
  end
  object BotonSalir: TButton
    Left = 328
    Top = 208
    Width = 75
    Height = 25
    Caption = 'Salir'
    TabOrder = 4
    OnClick = BotonSalirClick
  end
  object BotonAzul: TButton
    Left = 328
    Top = 96
    Width = 73
    Height = 25
    Caption = 'Azul'
    TabOrder = 2
    OnClick = BotonAzulClick
  end
  object BotonRestaurar: TButton
    Left = 328
    Top = 176
    Width = 73
    Height = 25
    Caption = 'Restaurar'
    TabOrder = 3
    OnClick = BotonRestaurarClick
  end
end
```

3.15 Patrones de diseño

Un patrón de diseño describe una solución a un problema en un contexto particular de tal forma que otros puedan reutilizar esta solución

Un patrón de diseño tiene cuatro elementos esenciales

- El **nombre del patrón**
  - Suele estar formado por una palabra o dos
  - Describe un problema de diseño
  - El nombre del patrón incrementa el vocabulario de diseño
  - Se debe buscar un buen nombre para cada patrón
- La **descripción del problema** al que se aplica el patrón
  - Se explica el problema y su contexto
  - Se describen problemas de diseño específicos
  - También se puede incluir una lista de las condiciones que se deben de cumplir antes de aplicar un patrón
- La **solución**
  - Describe los elementos que componen el diseño, así como sus relaciones, responsabilidades y colaboraciones
  - La solución no describe un diseño concreto o una implementación particular, ya que un patrón es como una plantilla que se puede aplicar a diferentes situaciones.
  - Un patrón muestra una descripción abstracta de un problema de diseño y como se organizan los elementos que resuelven el problema (clases y objetos en nuestro caso).
- Las **consecuencias**
  - Son los resultados y beneficios de aplicar el patrón
  - A través de las consecuencias se pueden evaluar las alternativas de diseño y estimar los costes y beneficios de aplicar un patrón
  - También pueden incluir aspectos sobre los lenguajes de programación
  - Dado que la reutilización de un patrón es un factor muy importante, en las consecuencias también se debe incluir el impacto sobre la flexibilidad, extensibilidad y portabilidad de los sistemas.
  - Un listado explícito de estas consecuencias ayuda a comprender y evaluar un patrón.



Catálogo de patrones de diseño

[Gamma 1995] [Cooper 2000] [Grand 1998] [Buschman 1996] [Buschman 2000]

Los patrones se pueden clasificar en tres grupos *[Gamma 1995]* **: de Creación, Estructurales y de Comportamiento. Aunque algunos autores añaden más patrones y más grupos.**

• **De creación (Creational).** Aportan una guía de cómo crear objetos, cuando su creación obliga a tomar decisiones. Estas decisiones se toman dinámicamente decidiéndose que clase se debe instanciar o que objetos deben delegar responsabilidades a otro objeto.

- **Abstract Factory**
- **Builder**
- **Factory Method**
- **Prototype**
- **Singleton**

• **Estructurales (Structural).** Describen como organizar diferentes tipos de objetos para que puedan trabajar juntos.

- **Adapter**
- **Bridge**
- **Composite**
- **Decorator**
- **Facade**
- **Flyweight**
- **Proxy**

• **Comportamiento (Behavioral).** Estos patrones se utilizan para organizar, gestionar y distribuir comportamientos.

- **Chain of Responsability**
- **Command**
- **Interpreter**
- **Iterator**
- **Mediator**
- **Memento**
- **Observer**
- **State**
- **Strategy**
- **Template Method**
- **Visitor**

Clasificación de patrones de diseño según su ámbito de aplicación

[Gamma 1995] [Grand 1998] [Grand 1999]

- **Los patrones se pueden clasificar en dos grupos: patrones de clases y patrones de objetos**

•De clases.

- **Factory Method**
- **Adapter**
- **Interpreter**
- **Template Method**

• De objetos.

- **Prototype**
- **Singleton**
- **Abstract Factory**
- **Builder**
- **Bridge**
- **Composite**
- **Decorator**
- **Facade**
- **Flyweight**
- **Proxy**
- **Chain of Responsability**
- **Command**
- **Iterator**
- **Mediator**
- **Memento**
- **Observer**
- **State**
- **Strategy**
- **Visitor**

Patrón Singleton (Único)

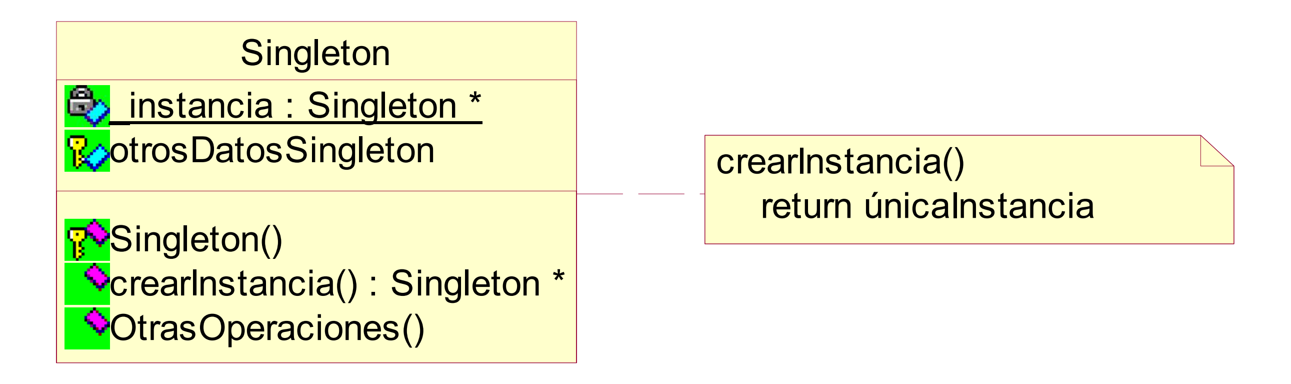
[Gamma 1995]

- **Propósito**
  - Garantiza que una clase sólo tenga una instancia
  - Proporciona un único punto de acceso global a ella
  - Se puede extender mediante herencia
- **Descripción del problema**
  - En algunas aplicaciones es importante garantizar que algunas clases tengan una sola instancia
  - ¿Cómo se puede asegurar que una clase tenga una única instancia y ésta sea fácilmente accesible?
  - Un objeto global está accesible por todos, pero no previene la creación de múltiples instancias

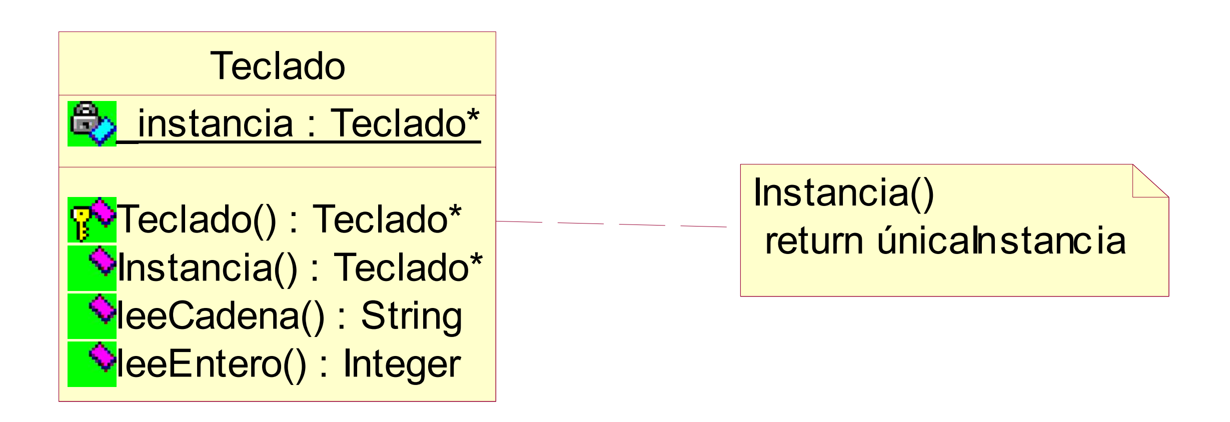
Patrón Singleton (Único)

[Gamma 1995]

- **Estructura de la solución**



- **Ejemplo**



Patrón Singleton (Único)

[Gamma 1995]

- ## Código ejemplo en C++ (Teclado.h)

```
// Teclado.h
// Contiene la clase Teclado
// Ejemplo del patrón de diseño Singleton (Único)
// Autor: Juan Manuel Cueva Lovelle
// Versión 1.0: 8 - Diciembre - 2002

#ifndef TECLADO_HPP
#define TECLADO_HPP

#include <iostream>
#include <string>
#include <stdlib.h> //para usar atoi()

using namespace std;

class Teclado
{
 private:
  static Teclado* _instancia;
 protected:
  Teclado (){cout<<"Creado teclado"<<endl;};
 public:
  static Teclado* Instancia();

  string leeCadena(){cout<<"Dame una cadena:";
                string cadena;
                cin>>cadena;
                return cadena;};

  int leeEntero(){cout<<"Dame un entero:";
                int entero;
                string cadena;
                cin>>cadena;
                //Conversión de cadena a entero
                //El método c_str() pasa objetos string a char*
                entero=atoi(cadena.c_str());
                return entero;};
};
#endif
```

Patrón Singleton (Único)

[Gamma 1995]

- **Código ejemplo en C++ (Teclado.cpp)**

```
// Teclado.cpp
// Contiene la implementación de la clase Teclado
// Ejemplo del patrón de diseño Singleton (Único)
// Autor: Juan Manuel Cueva Lovelle
// Versión 1.0: 8 - Diciembre - 2002

#include "Teclado.h"

Teclado* Teclado::_instancia=0;

Teclado* Teclado::Instancia(void){
                if (_instancia==0){
                        _instancia =new Teclado;
                }
                else
{cout<<"No se puede crear otro teclado, es el mismo"<<endl;};

 return _instancia;
}
```

Patrón Singleton (Único)

[Gamma 1995]

- **Código ejemplo en C++ (Prueba unitaria con C++ Builder)**

```
// PruebaPatronSingleton.cpp
// Ilustra la prueba unitaria de un patrón singleton
// Autor: Juan Manuel Cueva Lovelle
// Versión 1.0: 8 - Diciembre - 2002
// Compilado con C++ Builder 6.0 en modo Consola

#pragma hdrstop

#include "Teclado.h"
#include <conio>
#pragma argsused
int main(int argc, char* argv[])
{
        {
         Teclado* miTeclado = Teclado::Instancia();
         cout<< miTeclado->leeCadena()<<endl;
         cout << miTeclado->leeEntero()<<endl;
        }
         Teclado* otroTeclado = Teclado::Instancia();
// Sigue siendo el mismo teclado
         cout << otroTeclado->leeCadena()<<endl;
         cout << otroTeclado->leeEntero()<<endl;

        getch();
        return 0;
}
```

Patrón Singleton (Único)

[Gamma 1995]

- **Código ejemplo en C++ (Prueba unitaria con gcc)**

```
// PruebaPatronSingleton.cpp
// Ilustra la prueba unitaria de un patrón singleton
// Autor: Juan Manuel Cueva Lovelle
// Versión 1.0: 8 - Diciembre - 2002
// Compilado con g++ versión 2.96 (www.gnu.org)
// $ g++ -o PruebaPatronSingleton.out PruebaPatronSingleton.cpp Teclado.cpp
//------------------------------------------------------------------------------

#include "Teclado.h"

int main()
{
        {
         Teclado* miTeclado = Teclado::Instancia();
         cout<< miTeclado->leeCadena()<<endl;
         cout << miTeclado->leeEntero()<<endl;
        }
         Teclado* otroTeclado = Teclado::Instancia();
// Sigue siendo el mismo teclado
         cout << otroTeclado->leeCadena()<<endl;
         cout << otroTeclado->leeEntero()<<endl;

        return 0;
}
```

- **Ejecución de la prueba**

```
Creado teclado
Dame una cadena:qwerty
qwerty
Dame un entero:5
5
No se puede crear otro teclado, es el mismo
Dame una cadena:5
5
Dame un entero:555
555
```

3.16 Reutilización

[Meyer 97]

- Premisas
  - No reinventes la rueda, ¡herédala!
  - Usa y construye componentes
- Beneficios
  - Reducción de tiempos
  - Decremento del esfuerzo de mantenimiento
  - Fiabilidad
    - Cada desarrollador de un componente trabaja en su área de conocimiento
  - Eficiencia
  - Consistencia
  - Protección de la inversión en desarrollos
- La regla de la reutilización
  - Se debe ser consumidor de reutilización antes de ser un productor de reutilización
- ¿Qué se debe reutilizar?
  - Personal (formación)
  - Diseños y especificaciones (Patrones de Diseño)
  - Código fuente (componentes, herencia y genericidad)
  - Módulos abstractos (frameworks)
- Obstáculos para la reutilización
  - El síndrome NIA (No Inventado Aquí)
  - Impedimentos legales, comerciales o estratégicos
  - Acceso a a los fuentes de componentes y frameworks
  - Formatos de distribución de componentes (CORBA, ActiveX, ...)
  - Dilema reutilización-rehacer
  - Inercia del personal a los cambios

Precondiciones y post-condiciones

[Meyer 97]

- Factores de calidad
  - Corrección (Comprobación de aserciones)
  - Robustez (Manejo de excepciones)
- Asertos o aserciones
  - *son expresiones booleanas (con algunas extensiones) que manejan entidades software comprobando algunas propiedades que estas entidades pueden satisfacer en ciertas etapas de la ejecución del software.*
- Precondiciones y post-condiciones
  - Son dos aserciones asociadas a cada método de una clase
  - **Precondición**: *comprueba el estado de las propiedades que se deben cumplir* ***antes*** *de la ejecución del método.*
  - **Post-condición**: *comprueba el estado de las propiedades que se deben cumplir* ***después*** *de la ejecución del método.*
  - *El lenguaje Eiffel tiene dos secciones en cada método para precondiciones (**require**) y post-condiciones (**ensure**)*
  - Ejemplos: sea una Pila con los métodos *push* y *pop*
    - Precondiciones de *push*: la pila no puede estar llena
    - Post-condiciones de *push*: la pila no puede estar vacía
    - Precondiciones de *pop*: la pila no puede estar vacía
    - Post-condiciones de *pop*: la pila no puede estar llena

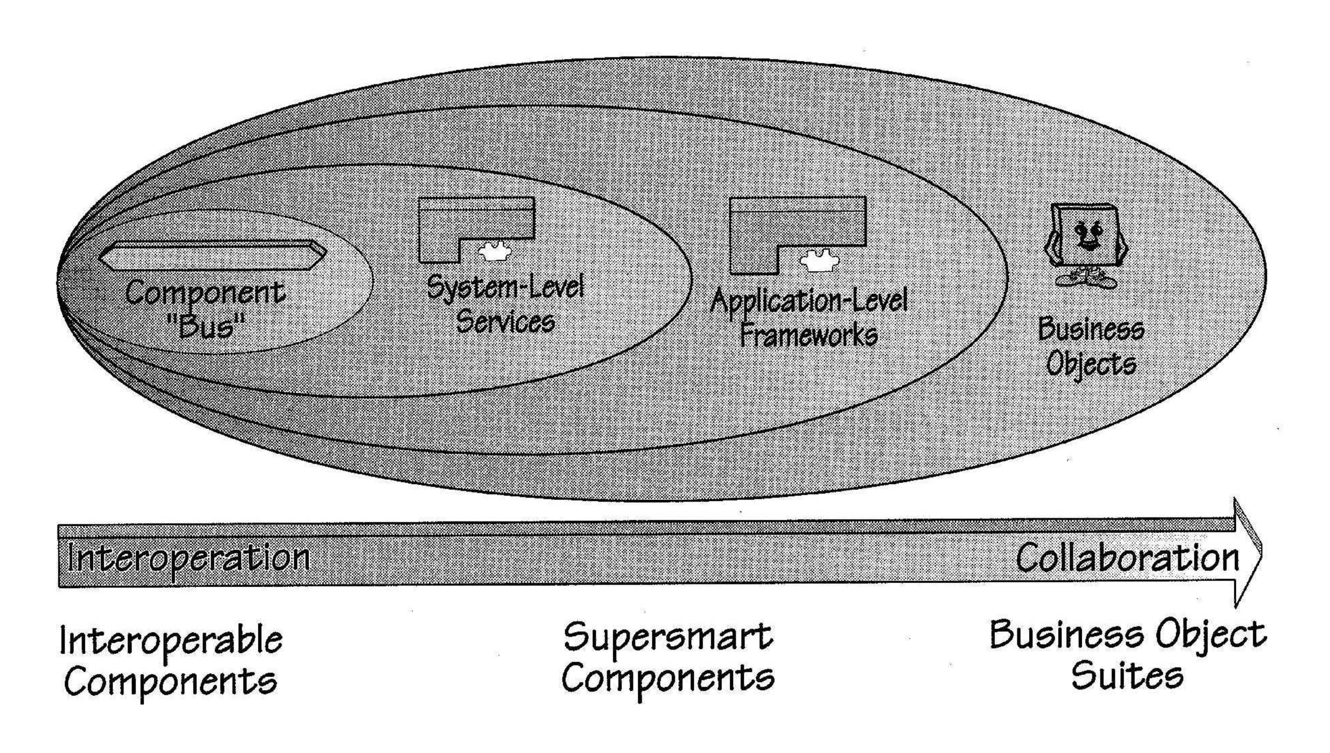
3.17 Diseño por contratos

[Meyer 97]

- Contratos para construir software fiable
  - Si se definen precondiciones y postcondiciones para cada método se establece un ***contrato*** que compromete al método (*suministrador de servicios*) y a los objetos que lo usan (*clientes*)
  - Las ***precondiciones*** comprometen al cliente: ellas definen las condiciones bajo las cuales el método funciona correctamente. Es una *obligación* para el cliente y un *beneficio* para el suministrador
  - Las ***postcondiciones*** comprometen a la clase: ellas definen las condiciones que deben asegurarse cuando finaliza el método. Es un *beneficio* para el cliente y una *obligación* para el suministrador.
  - Cuando se *rompe* el diseño por contratos se genera una *excepción*
- Diseño de precondiciones y postcondiciones
  - No deben ser ni demasiado grandes ni demasiado pequeñas. Deben comprobar aspectos estratégicos.
  - Deben aparecer en la documentación oficial (escenarios)
  - Se deben justificar en términos de la especificación
- Invariantes de clase
  - Son aserciones que capturan las propiedades semánticas profundas y la integridad de las restricciones que caracterizan la clase
  - Ejemplo de invariante de una Pila: *vacía=(contador=0)*
  - El lenguaje Eiffel tiene la sección *invariant* que se coloca antes del final de cada clase

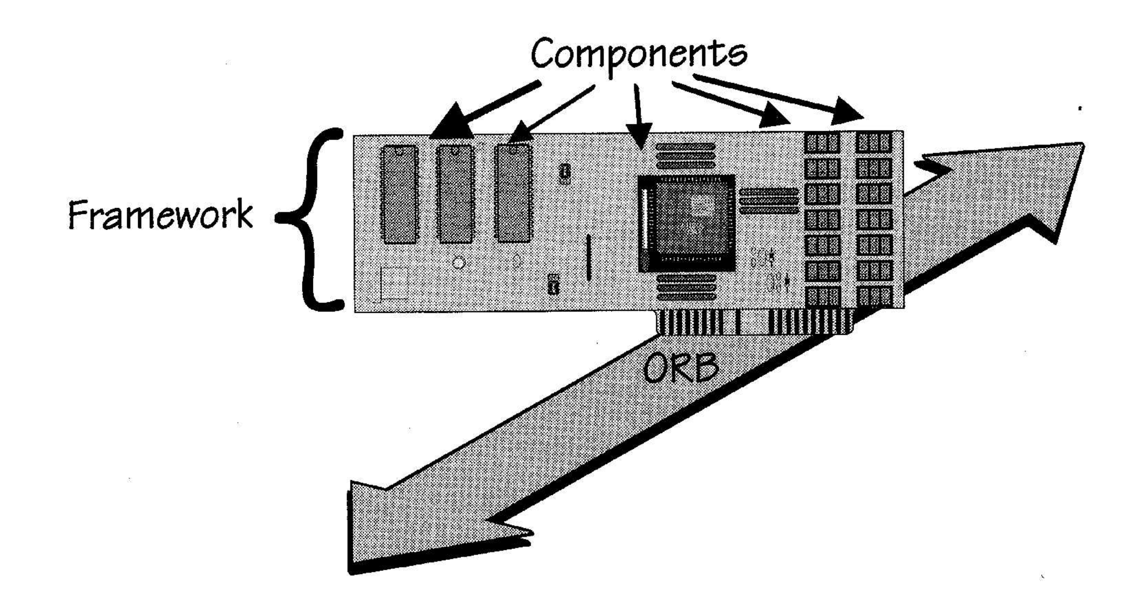
3.18 Estado actual de las TOO

- Bussiness Objects (objetos de negocio)
  - Son componentes que modelan completamente aplicaciones de un dominio determinado
- Frameworks



Frameworks

- Un framework (marco estructural) es un conjunto de bibliotecas de clases preparadas para la reutilización, que pueden utilizar a su vez componentes.
- Ejemplos
  - `www.junit.org` **(Pruebas de Software)**
  - `www.jhotdraw.org` **(Editor de dibujo)**
  - **.NET Framework (Base de la plataforma .NET)**



Analogía hardware de los frameworks *[Orfali 96]*

3.19 Perspectivas futuras de las Tecnologías Orientadas a Objetos

- Lenguajes O O
- Métodos y metodologías
- Distribución
- Componentes
- Ingeniería Web
- Sistemas Operativos O O
- Bases de datos Orientadas a Objetos
- Interfaces de usuario

Resumen

- **Beneficios del modelo de objetos**
  - Ayuda a explotar la potencia expresiva de los lenguajes de programación orientados a objetos, aunque algunas características del modelo de objetos no están soportadas.
  - Promueve la reutilización. No sólo de software aislado, sino también de diseños enteros (frameworks)
  - Produce sistemas que se construyen con formas intermedias estables
  - Facilita la comunicación con personas que son ajenas a la Informática
- **Aplicaciones de las Tecnologías de objetos**
  - Actualmente puede aplicarse a cualquier problema
  - Inconveniente: falta de personas con conocimientos adecuados

Preguntas de auto-evaluación (I)

1. ¿Qué se entiende por persistencia? **A)** Almacenar sólo los datos de un objeto en disco **B)** Un mecanismo por el cual los objetos siguen existiendo en el tiempo (por ejemplo se almacenan) o siguen existiendo en el espacio de direcciones de otra máquina **C)** Se dice cuando una clase insistentemente solicita al sistema operativo prioridad **D)** Se dice de los métodos que insisten en acceder a los datos **E)** Ninguno respuesta anterior es correcta o más de una lo es.
2. ¿Qué es una clase? **A)** Un caso particular de un objeto definido formalmente **B)** Un grupo de objetos **C)** Una colección de objetos **D)** una abstracción de un conjunto de objetos que comparten una estructura común y un comportamiento común **E)** Ninguna respuesta es correcta o más de una lo es.
3. ¿Qué es un objeto? **A)** Un ejemplar de una clase **B)** Una instancia de una clase **C)** Una variable declarada de un tipo que es una clase **D)** La estructura y comportamiento de objetos similares están definidos en su clase común **E)** Ninguna respuesta anterior es correcta o todas las anteriores son correctas.
4. La concurrencia en el modelo de objetos **A)** permite a diferentes objetos actuar al mismo tiempo **B)** Es la propiedad que distingue un objeto activo de uno que no está activo **C)** Permite construir modelos como un conjunto de objetos cooperativos, algunos de los cuales son activos y sirven así como centros de actividad independiente **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta.
5. El polimorfismo **A)** Se resuelve en tiempo de compilación **B)** Se produce cuando existen dos o más métodos con el mismo nombre pero con distinto número y/o tipo de parámetros **C)** No está disponible en C++ **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta
6. La sobrecarga **A)** Se realiza en tiempo de ejecución por el mecanismo de ligadura tardía **B)** El lenguaje C++ utiliza *métodos virtuales* y punteros a objetos para implementar la sobrecarga **C)** Es un mecanismo que permite a un método realizar distintas acciones al ser aplicado sobre distintos tipos de objetos que son instancias de una misma jerarquía de clases. **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta.
7. Se entiende por ORB **A)** las siglas de Object Request Broker **B)** El gestor de peticiones de objetos **C)** El mecanismo que permite a los objetos comunicarse con otros a través de la red **D)** Todas las respuestas anteriores son ciertas **E)** Ninguna respuesta anterior es cierta
8. De las precondiciones se puede decir **A)** Son aserciones asociadas a cada método de una clase **B)** Comprueba el estado de las propiedades que se deben cumplir antes de la ejecución del método **C)** En el diseño por contratos son una obligación para el cliente y un beneficio para el suministrador **D)** Todas las respuestas anteriores son ciertas **E)** Ninguna respuesta anterior es cierta
9. El manejo de excepciones se utiliza en **A)** Situaciones previstas y que ocurren frecuentemente **B)** Cuando falla un contrato en el diseño por contratos **C)** Es un caso particular de la genericidad **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta.
10. Un componente es **A)** Una entidad que se comercializa separada **B)** No es una aplicación completa **C)** Pueden combinarse de formas impredecibles **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta.

Preguntas de auto-evaluación (II)

11. Uno de los problemas de la herencia múltiple es la herencia repetida **A)** Ocurre cuando una clase hereda de dos o más clases que a su vez heredan de una clase que es un antepasado común a ambas **B)** Algunos lenguajes prohiben la herencia repetida (Eiffel) **C)** En C++ se permite la herencia repetida **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta
12. ActiveX es un tipo de componente que **A)** puede utilizarse en cualquier sistema operativo **B)** puede utilizarse por distintos lenguajes de programación **C)** puede convertirse en un Java Beans **D)** Todas las respuestas son correctas **E)** Ninguna respuesta anterior es correcta
13. Indicar cual de los siguientes lenguajes es un lenguaje orientado a objetos puro **A)** C++ **B)** ADA95 **C)** Object Pascal **D)** Java **E)** Ninguno es orientado a objetos puro.
14. El principio de modularidad denominado abierto – cerrado, se enuncia como **A)** Los módulos deben ser simultáneamente abiertos y cerrados **B)** Cuando un módulo responde a requisitos que cambian **C)** Cuando se emplean medios extraordinarios para mantener en una operación de un módulo **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta*1.*
15. La República Manzanera Vetustosa desea reconvertir el hípico del Molinón en una lechería, encargando un proyecto de software para la gestión del ganado vacuno para la producción de leche. Después de un análisis Orientado a Objetos concienzudo por parte la empresa Chapuzosa se ha generado la siguiente lista de clases para representar los objetos del dominio del problema: **Vaca, Toro, Novilla, Ganadero, Establo, Inseminar, Reproducir, Fecha, Ordeñar**,... Si se nos pide que asesoremos a Chapuzosa ¿Qué clases de las siguientes eliminaríamos por no representar objetos del dominio de problema? **A)** Toro, Novilla **B)** Ganadero **C)** reproducir, inseminar, ordeñar **D)** Todas son correctas **E)** Ninguna es correcta
16. La clase Vaca se ha diseñado de la forma que se refleja en el siguiente código C++

```
class Vaca {
    protected:
        float peso;
        Fecha nacimiento, ultima_inseminacion, ultimo_parto;
public:
        Vaca();
        void    pon_peso (float unpeso);
        float ver_peso();
        ...
    }
```

Respecto del diseño de los atributos de la clase Vaca se puede decir **A)** Quedan ocultos para los objetos de las clases que heredan de Vaca **B)** No permite a los objetos Vaca tener identidad **C)** No permite a los objetos Vaca tener estado **D)** Todas las respuestas anteriores son correctas **E)** Ninguna respuesta anterior es correcta

17. Las vacas están en establos, y en cada establo está un servidor que se encarga de manejar todos los objetos del establo. Usted como asesor/a recomienda **A)** comprar un Object Request Broker **B)** Seguir el estándar definido por OMG **C)** Organizar los establos con el paradigma de objetos distribuidos siguiendo CORBA **D)** Todos los consejos anteriores son válidos y compatibles entre sí **E)** Ningún consejo anterior es aceptable
18. Ante los éxitos alcanzados se desean introducir establos de otros animales que producen leche: Cabras, Ovejas, Buffalas, ... Se desea construir una clase genérica para describir los objetos que producen leche, usted aconsejaría **A)** Utilizar polimorfismo **B)** Usar un template **C)** Crear una clase abstracta de la que hereden las clases que producen leche **D)** Todos los consejos anteriores conducen a la genericidad **E)** Ninguna respuesta anterior es cierta
19. Si un programador de la empresa Chapuzosa construye una clase Bicho_raro que hereda de forma múltiple de Vaca, Cabra y Oveja. Suponiendo que se ha definido el atributo patas a cada uno de estas clases . ¿Cuántas patas tiene los objetos de la clase Bicho_raro? **A)** 4 **B)** 8 **C)** 12 **D)** No tiene patas **E)** Ninguna respuesta es válida.
20. Alguien ha contado a los de Chapuzosa que lo mejor es que usen patrones de diseño, usted les explica **A)** Que son un framework potente **B)** Representan clases abstractas puras **C)** Son soluciones a problemas en un contexto particular y que se pueden reutilizar **D)** Todas las respuestas anteriores son correctas **E)** Todas las respuestas anteriores son falsas

Respuestas de auto-evaluación

1. B)
2. D)
3. E)
4. D)
5. E)
6. E)
7. D)
8. D)
9. B)
10. D)
11. D)
12. E)
13. D)
14. A)
15. C)
16. B)
17. D)
18. D)
19. C)
20. C)

Lecturas recomendadas

- Se recomienda la lectura de los capítulos 2, 3 y 4 del libro de Booch ***Análisis y diseño orientado a objetos con aplicaciones***. 2ª Edición [Booch 1994]

- Otra lectura complementaria es la parte B y C (capítulos 3 al 18) del libro de Meyer ***Construcción de software orientado a objetos*** 2ª Edición [Meyer 1997]

- Se recomienda consultar sobre aspectos de la notación UML el libro de G. Booch *et al.* ***El lenguaje unificado de modelado*** [Booch 1999]

- Sobre la implementación en lenguaje C++ se recomienda el libro ***El lenguaje de programación C++.*** *Edición especial.* Addison Wesley, 2002. [Stroustrup 2000]

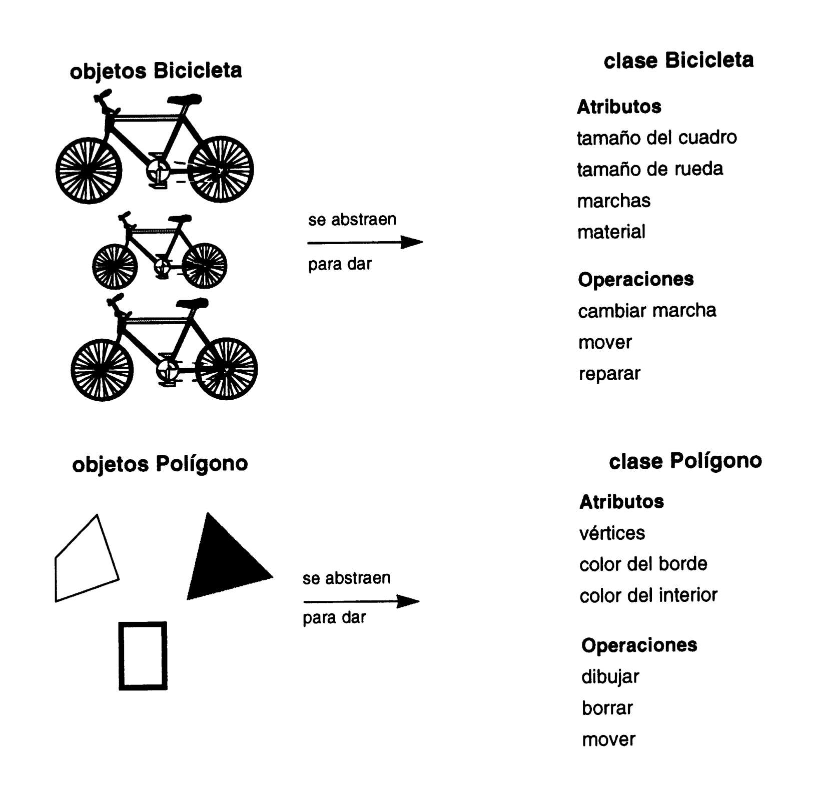
- Sobre buenos diseños orientados a objetos se recomienda la lectura del libro ***Patrones de diseño*** Addison-Wesley, 2003 [Gamma et al.1995]

Referencias

- [Arnold 1998] K. Arnold and J. Gosling. *The Java Programming Language. Second edition.* Addison Wesley, 1998
- [Baker 97] S. Baker.*CORBA Distributed Objects*. Addison-Wesley, 1997.
- [Booch 94] G.Booch. *Object-oriented analysis and design with applications.* Benjamin Cummings (1994). Versión castellana: *Análisis y diseño orientado a objetos con aplicaciones.* 2ª Edición. Addison-Wesley/ Díaz de Santos (1996).
- [Booch 1999] G. Booch, J. Rumbaugh, I. Jacobson. *The unified modeling language user guide.* Addison-Wesley (1999). Versión castellana *El lenguaje unificado de modelado.* Addison-Wesley (1999)
- [Buschman 1996] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, M. Stal. *Pattern-Oriented software architecture. Volume 1.* Wiley, 1996.
- [Buschman 2000] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, M. Stal. *Pattern-Oriented software architecture. Volume 2.* Wiley, 2000.
- [Charte 2001] F. Charte. *Kylix. Ed. Anaya, 2001.*
- [Chauvet 97] J-M Chauvet. *Corba, ActiveX y JavaBeans.* Ediciones Gestión 2000, 1997.
- [Coad 97] P. Coad, M. Mayfield. *Java design*. Prentice-Hall, 1997.
- [Cooper 2000] J.W. Cooper. *Java Design Patterns. Addison*-Wiley, 2000.
- [Cueva 93] Cueva Lovelle, J.M. García Fuente Mª P., López Pérez B., Luengo Díez Mª C., Alonso Requejo M. *Introducción a la programación estructurada y orientada a objetos con Pascal*. Distribuido por Ciencia-3 (1993).
- [Gamma 95] Gamma E. et al. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995. Edición en Castellano *Patrones de Diseño, Addison-Wesley, 2003.*
- [Grand 98] Grand M. *Patterns in Java. Volume 1.* Wiley, 1998.
- [Grand 99] Grand M. *Patterns in Java. Volume 2.* Wiley, 1999.
- [Jacobson 92] I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard. *Object-Oriented software Engineering. A use case driven Approach.* Addison-Wesley (1192)
- [Jacobson 99] I. Jacobson, G. Booch, J. Rumbaugh. *The unified software development process*. Addison-Wesley (1999).
- [Joyanes 96] L. Joyanes Aguilar. *Programación orientada a objetos. Conceptos, modelado, diseño y codificación en C++*. McGraw-Hill (1996).
- [Katrib 1996] M. Katrib Mora. *Programación Orientada a Objetos en C++. Ed. X view, México (1996)*
- [Khoshafian 90] S. Khoshafian & R. Abnous. *Object Orientation: Concepts, Languages, Databases, User Interfaces*. 1990, John Wiley & Sons, 0-471-51801-8.
- [Larman 98] C. Larman. *Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design*. Prentice-Hall (1998). Versión castellana: *UML y patrones. Introducción al análisis y diseño orientado a objetos*. Prentice-Hall (1999).
- [OMG] *www.omg.org*
- [Martin 96] J. Martin & J. J. Odell. *Object-Oriented Methods: Pragmatic Considerations*. 1996, Prentice Hall, 0-13-630864-3
- [Meyer 97] B. Meyer *Object-oriented software construction.* Second Edition. Prentice-Hall (1997). Versión castellana: *Construcción de software orientado a objetos*. Prentice-Hall (1998).
- [Orfali 96] R. Orfali, D. Harkey, J. Edwards. *The essential distributed objects. Survival guide.*Wiley (1996).
- [Orfali 97] R. Orfali, D. Harkey, J. Edwards. *Instant CORBA.*Wiley (1997).
- [Rational] Método Unificado y herramienta Rational/Rose en *www.rational.com*
- [Rumbaught 91] Rumbaught J., Blaha M., Premerlani W., Wddy F., Lorensen W. *Object-oriented modeling and design.* Prentice-Hall (1991). Versión castellana: *Modelado y diseño orientado a objetos. Metodología OMT*. Prentice-Hall (1996)
- [Stroustrup 1994] B. Stroustrup. *The design and evolution of C++*. Addison Wesley, 1994.
- [Stroustrup 2000] B. Stroustrup. *The C++ programming language. Special Edition.* Addison Wesley, 2000. Versión castellana *El lenguaje de programación C++. Edición especial.* Addison Wesley, 2002.
- [Szyperski 97] C. Szypersky. *Component software. Beyond Object-Oriented Programming.* Addison-Wesley (1997).
- [Taylor 92] D. A. Taylor. *Object-Oriented Information Systems: Planning and Implementation*. 1992, John Wiley & Sons, 0-471-54364-0
- [Tkach 94] D. Tkach & R. Puttick. *Object Technology in Application Development*, 1994, Benjamin/Cummings, 0-8053-2572-7
- [Wilkinson 95] N. M. Wilkinson. *Using CRC Cards. An Informal Approach to Object-Oriented Development*, 1995, SIGS BOOKS, 1-884842-07-0

Tema 4º
Clases y objetos

- Implementación de clases y objetos
- Compartición estructural y copia de objetos
- Interacción entre clases y objetos
- Construcción de clases y objetos de calidad
- Resumen



Implementación de clases y objetos

- Se implementará una clase Cola que se irá refinando en sucesivas versiones
- Todas las versiones se compilan y ejecutan correctamente pero tienen errores de diseño
- En este tema se repasan los conceptos del tema anterior desde un punto de vista práctico

Ejemplo de manejo de la clase cola en C++ con constructores y destructores por defecto

```
#include <iostream.h>
const Maximo=100;

// Declaración de la clase cola
// Define como opera una cola de enteros

class cola {
  int c[Maximo];       // Por defecto todo es privado
  int ultimo, primero; // Indices del array
public:
  void inicializa(void);
  void meter(int);
  int sacar(void);
};

void cola::inicializa()
{
  primero = ultimo = 0;
}

void cola::meter(int valor)
{
  if(ultimo==Maximo) // Precondición
   {
    cout << "La cola está llena\n";
    return;
   }
  ultimo++;
  c[ultimo] = valor;
}

int cola::sacar(void)
{
  if(ultimo == primero) //Precondición
   {
    cout << "La cola está vacía \n";
    return 0;
   }
  primero++;
  return c[primero];
}

int main(void)
{
  cola a, b;  // Se crean dos objetos de la clase cola

  a.inicializa();
  b.inicializa();
  a.meter(10);
  b.meter(19);
  a.meter(20);
  b.meter(1);
  cout << a.sacar() << " ";
  cout << a.sacar() << " ";
  cout << b.sacar() << " ";
  cout << b.sacar() << "\n";
}
```

Comentarios sobre el diseño de la versión 1 de la clase Cola

- El diseño del método inicializa()
  - Si se desea garantizar que inicializa() se ejecute cada vez que se crea un objeto esas operaciones deben estar dentro del constructor de la clase
- Se aconseja poner siempre constructores y destructores explícitos
  - Se puede colocar un simple mensaje
  - De esa forma sabemos siempre cuando se crean y se destruyen los objetos

Ejemplo de constructores y destructores en C++

```
// Versión 2
#include <iostream.h>
const Maximo=100;

class cola {
  int c[Maximo];
  int ultimo, primero;
public:
  cola(void); //Constructor de la clase. No puede devolver valores
  ~cola(void); //Destructor de la clase.
  void meter(int);
  int sacar(void);
};

cola::cola(void)  //Sustituye al método inicializa de la versión 1
{
  primero = ultimo = 0;
  cout << "La cola ha quedado inicializada\n";
}
cola::~cola(void) //Se ejecuta automáticamente al finalizar el programa
{
  cout << "La cola ha quedado destruída\n";
}
void cola::meter(int valor)
{
  if(ultimo==Maximo) //Precondición
   {
    cout << "La cola está llena\n";
    return;
   }
  ultimo++;
  c[ultimo] = valor;
}
int cola::sacar(void)
{
  if(ultimo == primero) //Precondición
   {
    cout << "La cola está vacía \n";
    return 0;
   }
  primero++;
  return c[primero];
}

int main(void)
{
  cola a, b;  // Se crean dos objetos de tipo cola
  a.meter(10);
  b.meter(19);
  a.meter(20);
  b.meter(1);
  cout << a.sacar() << " ";
  cout << a.sacar() << " ";
  cout << b.sacar() << " ";
  cout << b.sacar() << "\n";
  return 0;
}
```

Comentarios sobre el diseño de la versión 2 de la clase Cola

- Los objetos de clase Cola no están identificados
  - Si los objetos se almacenasen en disco no podríamos distinguir dos colas
- Se aconseja poner un atributo para identificar los objetos cola (versión 3)
- El atributo se pasa en el constructor
  - Un inconveniente es que no se garantiza que no se repitan identificaciones
  - Se puede hacer una versión con este atributo ***static*** y generado internamente por la propia clase

Ejemplo de constructores con parámetros en C++

```
// Versión 3

#include <iostream.h>
const Maximo=100;
class cola {
  int c[Maximo];
  int ultimo, primero;
  int identificacion; //Permite distinguir las colas
public:
  cola(int);   //Constructor de la clase con un parámetro
  ~cola(void); //Destructor de la clase.
  void meter(int);
  int sacar(void);
};

cola::cola(int id)  //Constructor con parámetro. Versión 3.
{
  primero = ultimo = 0;
  identificacion=id;
  cout << "La cola "<<identificacion<<" ha quedado inicializada"<<endl;
}
cola::~cola(void) //Se ejecuta automáticamente al finalizar el programa
{
  cout << "La cola "<<identificacion<<" ha quedado destruída"<<endl;
}
void cola::meter(int valor)
{
  if(ultimo==Maximo) //Precondición
   {
    cout << "La cola está llena\n";
    return;
   }
  ultimo++;
  c[ultimo] = valor;
}
int cola::sacar(void)
{
  if(ultimo == primero) //Precondición
   {
    cout << "La cola está vacía \n";
    return 0;
   }
  primero++;
  return c[primero];
}

int main(void)
{                  // Dos formas de paso de argumentos:
  cola a=cola(1); //Forma general
  cola b(2);      //Forma abreviada
  a.meter(10);
  b.meter(19);
  a.meter(20);
  b.meter(1);
  cout << a.sacar() << " ";
  cout << a.sacar() << " ";
  cout << b.sacar() << " ";
  cout << b.sacar() << endl; //Observar la ejecución
}
```

Comentarios sobre el diseño de la versión 3 de la clase Cola

- No se utiliza la modularidad de C++
  - Se separa en tres ficheros (versión 4)
    - `cola.hpp`. Donde se declaran las clases
    - `cola.cpp`. Donde se implementan los métodos
    - `prueba_cola_version_4.cpp.` Donde se realiza la prueba unitaria de la clase.
- Se implementa internamente con un array circular
- Se introduce un atributo `num_elementos` como auxiliar para manejar el array circular y para comprobar ciertas operaciones.
- Se introducen **postcondiciones**
  - La programación de postcondiciones se realiza con código redundante que verifica por partida doble la correción de las operaciones
- Se introducen métodos **selectores** e **iteradores**

```
// Capítulo Clases y Objetos: Ejercicio 4
// cola.hpp
//
// Ejemplo de métodos selector e iterador
//

// Versión 4
#ifndef COLA_HPP
#define COLA_HPP
//----------------------
#include <iostream.h>
//----------------------
const Maximo=3;

class cola {
  int c[Maximo];
  int ultimo, primero;
  int identificacion; //Permite distinguir las colas
  int num_elementos;
public:
  cola(int);   //Constructor de la clase con un parámetro
  ~cola(void); //Destructor de la clase.
  int estaVacia(void);//Método selector
  int estaLlena(void);//Método selector
  int longitud(void) {return num_elementos;};//Método selector
  void meter(int);  //Método modificador
  int sacar(void);  //Método modificador
  int ver_primero(void); //Método selector
  int ver_ultimo(void);   //Método selector
  int ver_identificacion(void){return identificacion;}; //Método selector
  int ver_posicion(int); //Método selector
  void mostrar_cola(void); //Método iterador
  void ver_array(void);//Método iterador utilizado solo para depurar
};
//------------------
#endif
```

```
// Capítulo Clases y Objetos: Ejercicio 4
// cola.cpp
// Implementación de la clase cola
// con un array circular

// Versión 4

#include "cola.hpp"

cola::cola(int id)  //Constructor con parámetro
{
  primero = ultimo = 0;
  identificacion=id;
  num_elementos=0;
  cout << "El objeto cola "<<identificacion<<" ha sido creado"<<endl;
}

cola::~cola(void) // Se ejecuta automáticamente al finalizar el bloque
{
  cout << "El objeto cola "<<identificacion<<" ha sido destruido"<<endl;
}

int cola::estaVacia(void)// Método selector
{
  if(num_elementos==0) return 1;
   else return 0;
}

int cola::estaLlena(void)// Método selector
{
  if(num_elementos==Maximo) return 1;
   else return 0;
}

void cola::meter(int valor) //Método modificador
// Precondición: La cola no tiene que estar llena
// Modifica: Almacena valor en la cola
// Postcondición: La cola tiene un elemento más
{
  if(estaLlena()) //Precondición
   {
    cout <<"No se puede introducir el "<< valor;
    cout <<" la cola "<<ver_identificacion()<<" está llena"<<endl;
    return;
   }
  int num_elementos_antes=longitud();

  cout<<"Introduciendo "<<valor<<" en la cola "<<ver_identificacion()<<endl;



  c[ultimo] = valor;
  num_elementos++;

  ultimo++;
  // Si se alcanza el final del array se vuelve al principio
  if (ultimo==Maximo) ultimo=0;

  //Postcondición
  int num_elementos_despues=longitud();
  if (num_elementos_despues!=num_elementos_antes+1)
    {
     cout <<"Error: La cola "<<ver_identificacion();
     cout<<" no tiene un elemento más"<<endl;
     return;
    }
}
```

```
int cola::sacar(void)
//Precondición: La cola no puede estar vacia
//Modifica: Devuelve y elimina de la cola el primer elemento
//Postcondición: La cola tiene un elemento menos
{
  if(estaVacia()) //Precondición
   {
    cout << "La cola "<<ver_identificacion()<<" está vacía \n";
    return -1;
   }
  int num_antes=longitud();


  int valor=c[primero];
  num_elementos--;
  primero++;
  // Si se alcanza el final del array se vuelve al principio
  if (primero==Maximo) primero=0;

  cout<<"Sacando "<<valor<<" de la cola "<<ver_identificacion()<<endl;


  int num_despues=longitud();
  if (num_antes!=num_despues+1) //Postcondición
   {
     cout<<"Error: La cola "<<ver_identificacion();
     cout<<" no tiene un elemento menos"<<endl;
     return -1;
    }
  return valor;
}

int cola::ver_primero(void)
{
  if(estaVacia()) //Precondición
   {
    cout << "La cola está vacía \n";
    return -1;
   }
  return c[primero];
}

int cola::ver_ultimo(void)
{
  if(estaVacia()) //Precondición
   {
    cout << "La cola está vacía \n";
    return 0;
   }

  if (ultimo>0) return c[ultimo-1];
   else return c[Maximo-1];
}
```

```
int cola::ver_posicion(int i)
{
  if(estaVacia()) //Primera precondición
   {
    cout << "La cola está vacía \n";
    return -1;
   }

  if (i>longitud()) //Segunda precondición
   {
    cout << "La cola solo tiene "<<longitud()<<endl;
    return -1;
   }
  if(primero+i-1<Maximo) return c[primero+i-1];
   else return c[primero+i-1-Maximo];
}

void cola::mostrar_cola(void)
{
  cout<<"La cola "<<ver_identificacion()<<" :";
  if (estaVacia())
   {
     cout<<"está vacía"<<endl;
     return;
    }
  for (int i=1; i<=longitud();i++)
     cout << ver_posicion(i)<<" ";
  cout <<endl;
}

void cola::ver_array(void)
 {
  for(int i=0;i<Maximo;i++)
  cout<<"c["<<i<<"]="<<c[i]<<endl;
  }
```

```
// Capítulo Clases y Objetos: Ejercicio 4
// prueba_cola_version_4.cpp
// Prueba unitaria de la clase cola
// Versión 4
#include "cola.hpp"
int main(void)
{
  cola a(1),b(2);
  a.meter(10);
  b.meter(19);
  a.meter(20);
  b.meter(1);
  a.mostrar_cola();
  a.ver_array();
  cout <<"La cola "<<a.ver_identificacion();
  cout <<" tiene "<<a.longitud()<<" elementos"<<endl;
  cout <<"El ultimo de la cola "<<a.ver_identificacion();
  cout <<" es "<<a.ver_ultimo()<<endl;
  cout <<"El primero de la cola "<<a.ver_identificacion();
  cout <<" es "<<a.ver_primero()<<endl;
  a.mostrar_cola();
  a.meter(500);
  cout <<"La cola "<<a.ver_identificacion();
  cout <<" tiene "<<a.longitud()<<" elementos"<<endl;
  a.mostrar_cola();
  a.meter(1000);
  a.mostrar_cola();
  a.ver_array();
  cout <<"El ultimo de la cola "<<a.ver_identificacion();
  cout <<" es "<<a.ver_ultimo()<<endl;
  cout <<"El primero de la cola "<<a.ver_identificacion();
  cout <<" es "<<a.ver_primero()<<endl;
  cout<<"Sacando elementos de la cola "<<b.ver_identificacion()<<endl;
  cout << b.sacar() << endl;
  cout << b.sacar() << endl;
  b.ver_array();
  b.mostrar_cola();
  b.meter(201);
  b.ver_array();
  b.meter(202);
  b.ver_array();
  b.meter(203);
  b.ver_array();
  b.mostrar_cola();
  cout<<"Primero "<<b.ver_primero()<<endl;
  cout<<"Ultimo "<<b.ver_ultimo()<<endl;
  b.sacar();
  b.mostrar_cola();
  b.ver_array();
  cout<<"Primero "<<b.ver_primero()<<endl;
  cout<<"Ultimo "<<b.ver_ultimo()<<endl;
  cout <<"La cola "<<b.ver_identificacion();
  cout <<" tiene "<<b.longitud()<<" elementos"<<endl;
  b.meter(204);
  b.mostrar_cola();
  b.ver_array();
  cout<<"Primero "<<b.ver_primero()<<endl;
  cout<<"Ultimo "<<b.ver_ultimo()<<endl;
  cout <<"La cola "<<b.ver_identificacion();
  cout <<" tiene "<<b.longitud()<<" elementos"<<endl;
}
```

Ejecución del Ejercicio 4 del capítulo Clases y Objetos

```
El objeto cola 1 ha sido creado
El objeto cola 2 ha sido creado
Introduciendo 10 en la cola 1
Introduciendo 19 en la cola 2
Introduciendo 20 en la cola 1
Introduciendo 1 en la cola 2
La cola 1 :10 20
c[0]=10
c[1]=20
c[2]=5234
La cola 1 tiene 2 elementos
El ultimo de la cola 1 es 20
El primero de la cola 1 es 10
La cola 1 :10 20
Introduciendo 500 en la cola 1
La cola 1 tiene 3 elementos
La cola 1 :10 20 500
No se puede introducir el 1000 la cola 1 está llena
La cola 1 :10 20 500
c[0]=10
c[1]=20
c[2]=500
El ultimo de la cola 1 es 500
El primero de la cola 1 es 10
Sacando elementos de la cola 2
Sacando 19 de la cola 2
19
Sacando 1 de la cola 2
1
c[0]=19
c[1]=1
c[2]=2640
La cola 2 :está vacía
Introduciendo 201 en la cola 2
c[0]=19
c[1]=1
c[2]=201
Introduciendo 202 en la cola 2
c[0]=202
c[1]=1
c[2]=201
Introduciendo 203 en la cola 2
c[0]=202
c[1]=203
c[2]=201
La cola 2 :201 202 203
Primero 201
Ultimo 203
Sacando 201 de la cola 2
La cola 2 :202 203
c[0]=202
c[1]=203
c[2]=201
Primero 202
Ultimo 203
La cola 2 tiene 2 elementos
Introduciendo 204 en la cola 2
La cola 2 :202 203 204
c[0]=202
c[1]=203
c[2]=204
Primero 202
Ultimo 204
La cola 2 tiene 3 elementos
El objeto cola 2 ha sido destruido
El objeto cola 1 ha sido destruido
```

Comentarios sobre el diseño de la versión 4 de la clase Cola

- Siguiendo el principio de ocultación de la información se diseña una nueva clase Cola
  - La implementación interna es con punteros (parte privada)
  - Se mantienen los mismos métodos públicos
- Ahora el destructor no es un simple comentario
  - S e debe liberar memoria
  - Se vacían los objetos cola antes de destruirlos

```
// Capítulo Clases y Objetos: Ejercicio 5
// cola.hpp
//
// Clase cola implementada con punteros
//

// Versión 5


#ifndef COLA_HPP
#define COLA_HPP
//----------------------
#include <iostream.h>
//----------------------


struct nodo {
        int info;
        struct nodo *sig;
        };

typedef struct nodo NODO;

class cola {
  NODO *ultimo, *primero;
  int identificacion; //Permite distinguir las colas
  int num_elementos;
public:
  cola(int);   //Constructor de la clase con un parámetro
  ~cola(void); //Destructor de la clase.
  int estaVacia(void);//Método selector
  int longitud(void) {return num_elementos;};//Método selector
  void meter(int);  //Método modificador
  int sacar(void);  //Método modificador
  int ver_primero(void); //Método selector
  int ver_ultimo(void);   //Método selector
  int ver_identificacion(void){return identificacion;}; //Método selector
  int ver_posicion(int); //Método selector
  void mostrar_cola(void); //Método iterador
};
//-------------------
#endif
```

```
// Capítulo Clases y Objetos: Ejercicio 5
// cola.cpp
// Implementación de la clase cola
// con estructuras dinámicas de datos

// Versión 5

#include "cola.hpp"

cola::cola(int id)  //Constructor con parámetro
{
  primero = NULL;
  ultimo = NULL;
  identificacion=id;
  num_elementos=0;
  cout << "El objeto cola "<<identificacion<<" ha sido creado"<<endl;
}

cola::~cola(void) // Es necesario vaciarla
{
  int i;
// Vaciar la cola para liberar memoria
  while (!estaVacia()) sacar();
  cout << "El objeto cola "<<identificacion<<" ha sido destruido"<<endl;
}

int cola::estaVacia(void)// Método selector
{
  if(num_elementos==0) return 1;
   else return 0;
}

void cola::meter(int valor) //Método modificador
// Modifica: Almacena valor en la cola
// Postcondición: La cola tiene un elemento más
{
  int num_elementos_antes=longitud();
  cout<<"Introduciendo "<<valor<<" en la cola "<<ver_identificacion()<<endl;
  NODO *nuevo;
  nuevo=new NODO;  //Reserva dinámica de memoria para nuevo
  nuevo->info= valor;
  nuevo->sig=NULL;
  num_elementos++;

  if (ultimo==NULL)  //Caso de cola vacia
             {
     ultimo=nuevo;
     primero=nuevo;
    }
   else       //Caso en el que la cola ya tiene elementos
    {
     ultimo->sig=nuevo;
     ultimo=nuevo;
    }
  //Postcondición
  int num_elementos_despues=longitud();
  if (num_elementos_despues!=num_elementos_antes+1)
    {
     cout <<"Error: La cola "<<ver_identificacion();
     cout<<" no tiene un elemento más"<<endl;
     return;
    }
}
```

```
int cola::sacar(void)
//Precondición: La cola no puede estar vacia
//Modifica: Devuelve y elimina de la cola el primer elemento
//Postcondición: La cola tiene un elemento menos
{
  if(estaVacia()) //Precondición
   {
    cout << "La cola "<<ver_identificacion()<<" está vacía \n";
    return -1;
   }
  int num_antes=longitud();

  NODO *p;
  int valor=primero->info;
  p=primero;
  primero=primero->sig;
  delete p;    //Elimina p de la memoria dinámica
  if (primero==NULL) ultimo=NULL;
  num_elementos--;

  cout<<"Sacando "<<valor<<" de la cola "<<ver_identificacion()<<endl;

  int num_despues=longitud();
  if (num_antes!=num_despues+1) //Postcondición
   {
     cout<<"Error: La cola "<<ver_identificacion();
     cout<<" no tiene un elemento menos"<<endl;
     return -1;
    }
  return valor;
}

int cola::ver_primero(void)
{
  if(estaVacia()) //Precondición
   {
    cout << "La cola está vacía \n";
    return -1;
   }
  return primero->info;
}

int cola::ver_ultimo(void)
{
  if(estaVacia()) //Precondición
   {
    cout << "La cola está vacía \n";
    return 0;
   }
  return ultimo->info;
}
```

```
int cola::ver_posicion(int i)
{
  if(estaVacia()) //Primera precondición
   {
    cout << "La cola está vacía \n";
    return -1;
   }

  if (i>longitud()) //Segunda precondición
   {
    cout << "La cola solo tiene "<<longitud()<<endl;
    return -1;
   }

  NODO *p;
  p=primero;
  for (int j=1;j<i;j++) p=p->sig;
  return p->info;
}

void cola::mostrar_cola(void)
{
  cout<<"La cola "<<ver_identificacion()<<" :";
  if (estaVacia())
   {
     cout<<"está vacía"<<endl;
     return;
    }
  for (int i=1; i<=longitud();i++)
     cout << ver_posicion(i)<<" ";
  cout <<endl;
}
```

```
// Capítulo Clases y Objetos: Ejercicio 5
// c_o_e5.cpp
// Prueba unitaria de la clase cola
// Versión 5

#include "cola.hpp"
int main(void)
{
  cola a(1),b(2);

  a.meter(10);
  b.meter(19);
  a.meter(20);
  b.meter(1);
  a.mostrar_cola();
  cout <<"La cola "<<a.ver_identificacion();
  cout <<" tiene "<<a.longitud()<<" elementos"<<endl;
  cout <<"El ultimo de la cola "<<a.ver_identificacion();
  cout <<" es "<<a.ver_ultimo()<<endl;
  cout <<"El primero de la cola "<<a.ver_identificacion();
  cout <<" es "<<a.ver_primero()<<endl;
  a.mostrar_cola();
  a.meter(500);
  cout <<"La cola "<<a.ver_identificacion();
  cout <<" tiene "<<a.longitud()<<" elementos"<<endl;
  a.mostrar_cola();
  a.meter(1000);
  a.mostrar_cola();
  cout <<"El ultimo de la cola "<<a.ver_identificacion();
  cout <<" es "<<a.ver_ultimo()<<endl;
  cout <<"El primero de la cola "<<a.ver_identificacion();
  cout <<" es "<<a.ver_primero()<<endl;
  cout<<"Sacando elementos de la cola "<<b.ver_identificacion()<<endl;
  cout << b.sacar() << endl;
  cout << b.sacar() << endl;
  b.mostrar_cola();
  b.meter(201);
  b.meter(202);
  b.meter(203);
  b.mostrar_cola();
  cout<<"Primero "<<b.ver_primero()<<endl;
  cout<<"Ultimo "<<b.ver_ultimo()<<endl;
  b.sacar();
  b.mostrar_cola();
  cout<<"Primero "<<b.ver_primero()<<endl;
  cout<<"Ultimo "<<b.ver_ultimo()<<endl;
  cout <<"La cola "<<b.ver_identificacion();
  cout <<" tiene "<<b.longitud()<<" elementos"<<endl;
  b.meter(204);
  b.mostrar_cola();
  cout<<"Primero "<<b.ver_primero()<<endl;
  cout<<"Ultimo "<<b.ver_ultimo()<<endl;
  cout <<"La cola "<<b.ver_identificacion();
  cout <<" tiene "<<b.longitud()<<" elementos"<<endl;
}
```

Ejecución del ejercicio 5 del capítulo Clases y Objetos

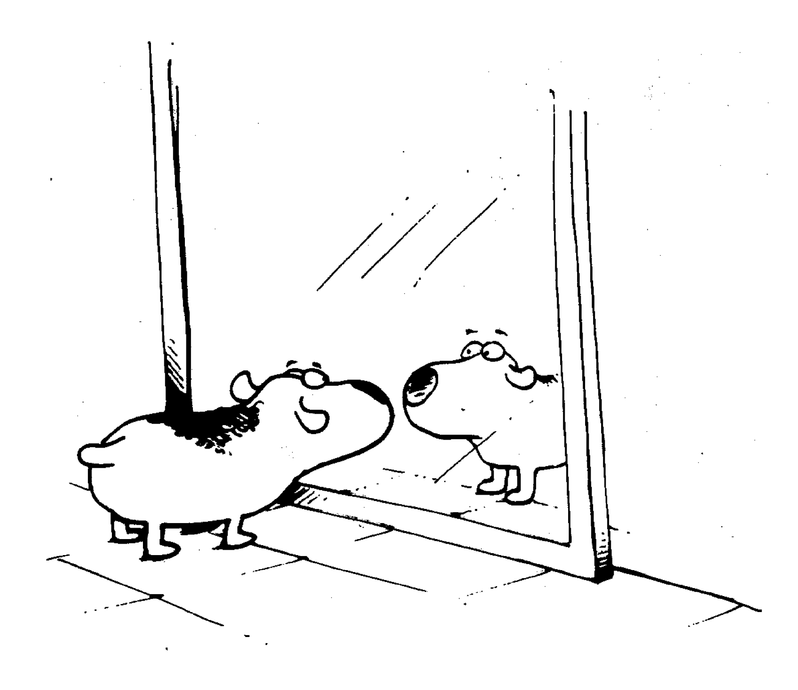
```
El objeto cola 1 ha sido creado
El objeto cola 2 ha sido creado
Introduciendo 10 en la cola 1
Introduciendo 19 en la cola 2
Introduciendo 20 en la cola 1
Introduciendo 1 en la cola 2
La cola 1 :10 20
La cola 1 tiene 2 elementos
El ultimo de la cola 1 es 20
El primero de la cola 1 es 10
La cola 1 :10 20
Introduciendo 500 en la cola 1
La cola 1 tiene 3 elementos
La cola 1 :10 20 500
Introduciendo 1000 en la cola 1
La cola 1 :10 20 500 1000
El ultimo de la cola 1 es 1000
El primero de la cola 1 es 10
Sacando elementos de la cola 2
Sacando 19 de la cola 2
19
Sacando 1 de la cola 2
1
La cola 2 :está vacía
Introduciendo 201 en la cola 2
Introduciendo 202 en la cola 2
Introduciendo 203 en la cola 2
La cola 2 :201 202 203
Primero 201
Ultimo 203
Sacando 201 de la cola 2
La cola 2 :202 203
Primero 202
Ultimo 203
La cola 2 tiene 2 elementos
Introduciendo 204 en la cola 2
La cola 2 :202 203 204
Primero 202
Ultimo 204
La cola 2 tiene 3 elementos
El objeto cola 2 ha sido destruido
El objeto cola 1 ha sido destruido
```

Compartición estructural

- *Se produce cuando la identidad de un objeto recibe un alias a través de un segundo nombre*
- *El mismo objeto tiene dos identificadores*
- *En lenguajes como C++ se permite el uso de alias o referencias*
- *No es aconsejable el uso de alias o de referencias que no sean parámetros de funciones*

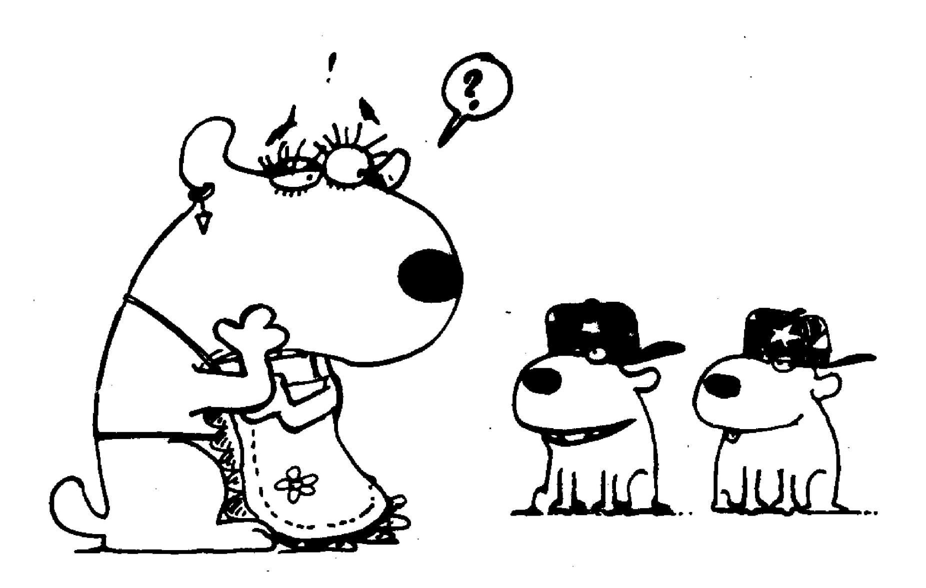
Ejemplo de referencias independientes en C++

```
class Persona
 {…};
Persona francisco;
Persona &paco=francisco; //paco y francisco
                         //son el mismo objeto
```



Copia de objetos

- *Se pretende obtener dos objetos con identificadores distintos pero con estados iguales*
- *En C++ no debe utilizarse el operador asignación (=) para crear copias de objetos, pues para objetos cuyo estado involucra a punteros a otros objetos, sólo se copia el puntero pero no a lo que apunta el puntero.*
- *Las soluciones son:*
  – Añadir a la clase constructores de copia
  – Sobrecargar el operador asignación en la clase para que realice la copia correctamente
- *Para determinar si dos objetos son iguales se deben sobrecargar los operadores de comparación == y !=*



Constructores de copia por defecto en C++

• Un constructor de copia es un constructor de la clase que se utiliza para copiar objetos de dicha clase
• C++ incorpora genera automáticamente un constructor de copia defecto (también denominado *constructor de copia por omisión*)
• Sólo se generará un constructor de copia por defecto si no se ha declarado un constructor de copia
• El constructor de copia por defecto se utiliza automáticamente en los siguientes casos:
  • En una asignación entre objetos de la misma clase
  • Cuando se pasa por valor un objeto de la clase
  • Cuando una función devuelve un objeto de la clase

```
// Capítulo Clases y Objetos: Ejercicio 6
// c_o_e6.cpp
// Uso del constructor de copia por defecto
// Versión 6
#include "cola.hpp"
int main(void)
{
  cola a(1),b(2);

  a.meter(10);
  a.meter(20);
  a.mostrar_cola();
  b=a;// la asignación utiliza el constructor de copia por defecto
  b.mostrar_cola();
  cout <<"La cola "<<b.ver_identificacion();
  cout <<" tiene "<<b.longitud()<<" elementos"<<endl;
  cout <<"El ultimo de la cola "<<b.ver_identificacion();
  cout <<" es "<<b.ver_ultimo()<<endl;
  cout <<"El primero de la cola "<<b.ver_identificacion();
  cout <<" es "<<b.ver_primero()<<endl;
}
```

Ejecución

```
El objeto cola 1 ha sido creado
El objeto cola 2 ha sido creado
Introduciendo 10 en la cola 1
Introduciendo 20 en la cola 1
La cola 1 :10 20
La cola 1 :10 20
La cola 1 tiene 2 elementos
El ultimo de la cola 1 es 20
El primero de la cola 1 es 10
El objeto cola 1 ha sido destruido
El objeto cola 1 ha sido destruido
```

Constructores de copia en C++

• Un constructor de copia es un constructor de la clase que se utiliza para copiar objetos de dicha clase
• Los constructores de copia son de la forma: `C(C&)` o `C(const C&)` donde C es el nombre de la clase.

```
// Capítulo Clases y Objetos: Ejercicio 7
// cola.hpp
//
// Ejemplo con constructor de copia y sobrecarga de operadores
//

// Versión 7
#ifndef COLA_HPP
#define COLA_HPP
//----------------------
#include <iostream.h>
//----------------------

struct nodo {
        void *info; //Cola genérica de punteros a void
        struct nodo *sig;
        };

typedef struct nodo NODO;

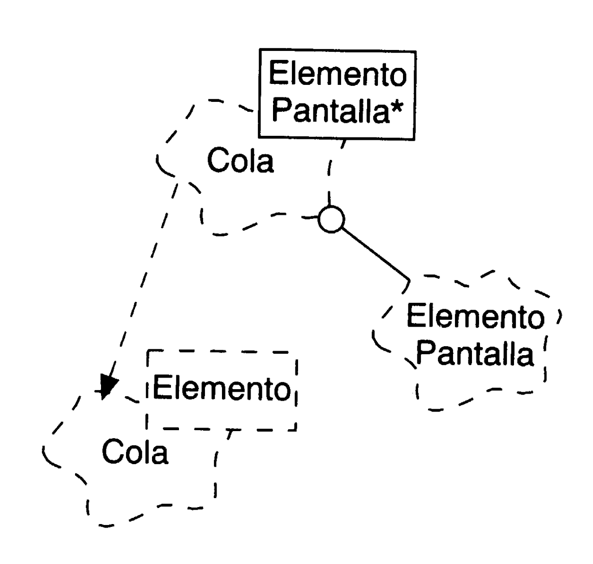
class cola {
  NODO *ultimo, *primero;
  int identificacion; //Permite distinguir las colas
  int num_elementos;
public:
  cola(int=0);   //Constructor de la clase con un parámetro por defecto
  cola(const cola&); //Constructor de copia
  cola& operator = (const cola&); // Sobrecarga de la asignación
  int operator == (const cola&); // Sobrecarga del operador comparación ==
  int operator != (const cola&); // Sobrecarga del operador comparación !=
  ~cola(void); //Destructor de la clase.
  int estaVacia(void);//Método selector
  int longitud(void) {return num_elementos;};//Método selector
  void meter(const void*);  //Método modificador
  const void* sacar(void);  //Método modificador
  const void* ver_primero(void); //Método selector
  const void* ver_ultimo(void);   //Método selector
  int ver_identificacion(void){return identificacion;}; //Método selector
  const void* ver_posicion(int); //Método selector
};
//-------------------
#endif
```

Genericidad

- La genericidad es soportada por lenguajes OO como C++ y Eiffel mediante el uso de clases parametrizadas ("plantillas" o "templates")
- Las clases parametrizadas permiten escribir una plantilla que se pueden aplicar a varios casos que difieren solamente en los tipos de los parámetros
- Una clase parametrizada no puede tener instancias a menos que antes se la instancie indicando los parámetros
- Usando el ejemplo en C++ para definir dos objetos cola

```
Cola<int> colaEnteros;
Cola<ElementoPantalla*>colaElementos;
```

- Las clases parametrizadas son seguras respecto al conrol de tipos que se determina en tiempo de compilación



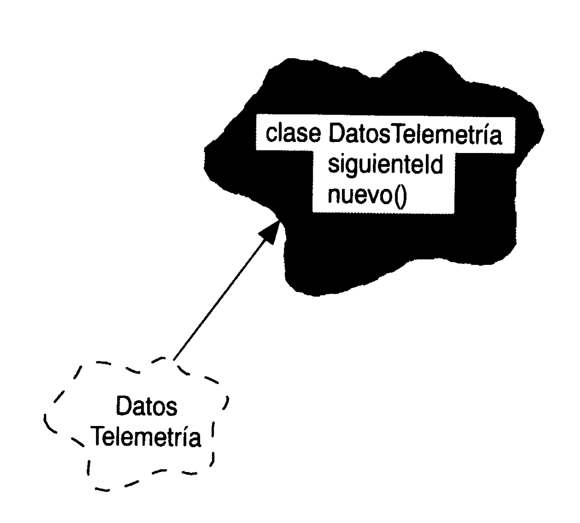
Notación de Booch

Ejemplo en C++

```
template<class Elemento>
class Cola{
public:
            Cola();
            Cola(const Cola<Elemento>&);
            virtual ~Cola();
virtual Cola<Elemento>& operator=(const Cola<Elemento>&);
virtual int operator==(const Cola<Elemento>&)const;
int operator !=(const Cola<Elemento>&) const;
virtual void borrar();
virtual void anadir(const Elemento&);
. . .
}
```

Metaclases

- *Una metaclase es una clase cuyas instancias son clases*
- Los lenguajes Smalltalk y CLOS soportan metaclases
- Los lenguajes C++, Object Pascal, Ada, Eiffel no lo soportan
- El problema es que se deben crear clases en tiempo de ejecución
- La notación de Booch da soporte a las metaclases con una nube gris y una flecha gris.



Metaclases.

Interacción entre clases y objetos

- **Relaciones entre clases y objetos**
  - Las clases y objetos son conceptos separados pero muy relacionados entre sí
  - Todo objeto es instancia de alguna clase
  - Toda clase tiene cero o más instancias
  - ***Las clases son estáticas***. Su existencia y semántica están fijadas antes de la ejecución del programa. La única excepción son los lenguajes que usan metaclases
  - ***Los objetos son dinámicos***. Se crean y se destruyen en tiempo de ejecución, un caso particular son los objetos persistentes
- **El papel de clases y objetos en análisis y diseño**. Durante el análisis y las primeras etapas del diseño el desarrollador tiene dos tareas principales:
  - *Identificar las clases y objetos que forman el vocabulario del dominio del problema (**abstracciones clave**)*
  - *Idear las estructuras por las que conjuntos de objetos trabajan juntos para lograr los comportamientos que satisfacen los requisitos del problema (**mecanismos**)*

Construcción de clases y objetos de calidad

- Medida de la calidad de una abstracción
  - Acoplamiento *es la medida de la fuerza de la asociación establecida por una conexión entre un módulo y otro*
    - Acoplamiento entre módulos débil reduce la complejidad
    - Acoplamiento entre clases y objetos versus herencia (acoplamiento fuerte)
  - Cohesión *mide el grado de conectividad entre los elementos de un sólo módulo y en DOO la de los elementos de una clase*
    - La peor: **cohesión por coincidencia** *(las abstracciones de un mismo módulo o clase no tienen ninguna relación)*
    - La mejor: **cohesión funcional** *(las abstracciones trabajan conjuntamente para proporcionar un comportamiento determinado)*
  - Suficiencia: *la clase o módulo captura suficientes características de la abstracción como para permitir una interacción significativa y eficiente*
  - Completud (estado completo o plenitud): *El interfaz de la clase o módulo captura todas las características significativas de la abstracción*
  - Operaciones primitivas: *Sólo se implementan en la clase o el módulo aquellas operaciones cuya implementación eficiente sólo se produce si se tiene acceso a la representación interna de la abstracción*

Construcción de clases y objetos de calidad (II)

- Selección de operaciones
  - Semántica funcional: *Seleccionar el interfaz de una clase es complejo. Un buen diseñador debe llegar a un equilibrio entre subdividir demasiado en subclases y dejar clases o módulos demasiado grandes*
  - Semántica espacial: *Especifica la cantidad de espacio de almacenamiento de cada operación*
  - Semántica temporal: *Especifica la semántica de la concurrencia en términos de sincronización.*
- Elección de relaciones
  - Colaboraciones
    - Los métodos de una clase tan sólo deben depender de la propia clase
    - Un método sólo debe enviar mensajes a objetos de un número limitado de clases
    - La jerarquía de herencias puede ser en forma de bosque (debilmente acopladas) y árboles (acoplamiento fuerte). La jerarquía de herencias elegida depende del tipo de problema
  - Mecanismos y visibilidad
    - Las relaciones entre objetos plantean el diseño de mecanismos que reflejan como los objetos interactúan
    - Es útil definir como un objeto es visible para otros

Construcción de clases y objetos de calidad (III)

- Elección de implementaciones
  - Representación
    - *La representación interna de una clase es un secreto encapsulado de la abstracción*
    - *La elección de la representación es difícil y no unívoca*
    - *La modificación de la representación interna no debe violar ninguno de los contratos con los clientes de la clase*
  - Empaquetamiento
    - *Elegir las clases y objetos que se empaquetan en cada módulo*
    - *Los requisitos de ocultación y visibilidad entre módulos suelen guiar las decisiones de diseño*
    - *Generalmente los módulos tienen cohesión funcional y están débilmente acoplados entre sí*
    - *Hay factores no técnicos que influyen en la definición de módulos: reutilización, seguridad y documentación.*

Resumen

- Un objeto tiene estado, comportamiento e identidad
- La estructura y comportamiento de objetos similares están definidos en su clase común
- El estado de un objeto abarca todas las propiedades (normalmente estáticas) del mismo más los valores actuales (normalmente dinámicos) de cada una de esas propiedades
- El comportamiento es la forma que un objeto actúa y reacciona en términos de sus cambios de estado y paso de mensajes
- La identidad es la propiedad de un objeto que lo distingue de todos los demás objetos
- Los dos tipos de jerarquías de objetos son relaciones de asociación y agregación
- Una clase es un conjunto de objetos que comparten una estructura y comportamiento comunes
- Los seis tipos de jerarquías de clase son las relaciones de asociación, herencia, agregación, "uso", instanciación, clases parametrizadas y relaciones de metaclase
- Las abstracciones clave son las clases y objetos que forman el vocabulario del dominio del problema
- Un mecanismo es una estructura por la que un conjunto de objetos trabajan juntos para ofrecer un comportamiento que satisfaga algún requisito del problema
- La calidad de una abstracción puede medirse por su acoplamiento, cohesión, suficiencia, completud (estado completo) y las operaciones primitivas

Autoevaluación (I)

- Un iterador es **A)** Una operación que altera el estado de un objeto **B)** Una operación que accede al estado de un objeto, pero no altera este estado **C)** Una operación que permite acceder a todas las partes de un objeto en algún orden perfectamente establecido **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta.
- Un selector es **A)** Una operación que altera el estado de un objeto **B)** Una operación que accede al estado de un objeto, pero no altera este estado **C)** Una operación que permite acceder a todas las partes de un objeto en algún orden perfectamente establecido **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta.
- Un modificador es **A)** Una operación que altera el estado de un objeto **B)** Una operación que accede al estado de un objeto, pero no altera este estado **C)** Una operación que permite acceder a todas las partes de un objeto en algún orden perfectamente establecido **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta.
- Un constructor es **A)** Una operación que libera el estado de un objeto y/o destruye el propio objeto **B)** Una operación que accede al estado de un objeto, pero no altera este estado **C)** Una operación que permite acceder a todas las partes de un objeto en algún orden perfectamente establecido **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta.
- Un destructor es **A)** Una operación que altera el estado de un objeto **B)** Una operación que accede al estado de un objeto, pero no altera este estado **C)** Una operación que permite acceder a todas las partes de un objeto en algún orden perfectamente establecido **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta.
- El tiempo de vida de un objeto **A)** Se extiende desde que lo crea un constructor hasta que se destruye por un destructor **B)** En lenguajes con recolección de basura se extiende desde que se crea hasta que todas las referencias a el objeto se han perdido **C)** Si el objeto es persistente el tiempo de vida transciende a la vida del programa que lo crea **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta.
- Relaciones entre clases y objetos **A)** Las clases en C++ son estáticas **B)** Los objetos en C++ son dinámicos **C)** En C++ todo objeto es instancia de alguna clase **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta.
- Un constructor de una clase en C++ **A)** Siempre existe implícita o explícitamente **B)** Es una operación que accede al estado de un objeto, pero no altera este estado **C)** Una operación que permite acceder a todas las partes de un objeto en algún orden perfectamente establecido **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta.
- Un destructor **A)** Es una operación que no existe en los lenguajes con recolección de basura **B)** No existe en Java **C)** Siempre existe en C++ **D)** Todas las afirmaciones anteriores son correctas **E)** Ninguna respuesta anterior es correcta

Autoevaluación (II)

- Sea el siguiente fragmento de código en C++ que esta en un módulo denominado `avion.h`

```
class Ala {...}; // Los ... indican que se ha implementado la clase
class Motor {...};
class TrenDeAterrizaje{...};
class Cabina{...};
class Avion {
     Ala *a1, *a2;
     Motor *m1, *m2;
     TrenDeAterrizaje *t;
     Cabina *c;
     ...}
```

Se puede decir **A)** Que la clase Avion hereda de las anteriores **B)** Que `Ala` es un método virtual de la clase Avion **C)** Que la clase `Avion` es una agregación **D)** Todas las respuestas anteriores son correctas **E)** Todas las respuestas son falsas.

- Utilizando la clase Avion anterior se escribe el siguiente fragmento de código:

```
#include  "avion.h"
main ()
 {
  Avion A310;
  ...
 }
```

Se puede afirmar **A)** La instrucción `Avion A310;` es una llamada a un método constructor **B)** A310 es un objeto **C)** A310 se crea en tiempo de ejecución **D)** Todas las respuestas anteriores son correctas **E)** Todas las respuestas anteriores son falsas.

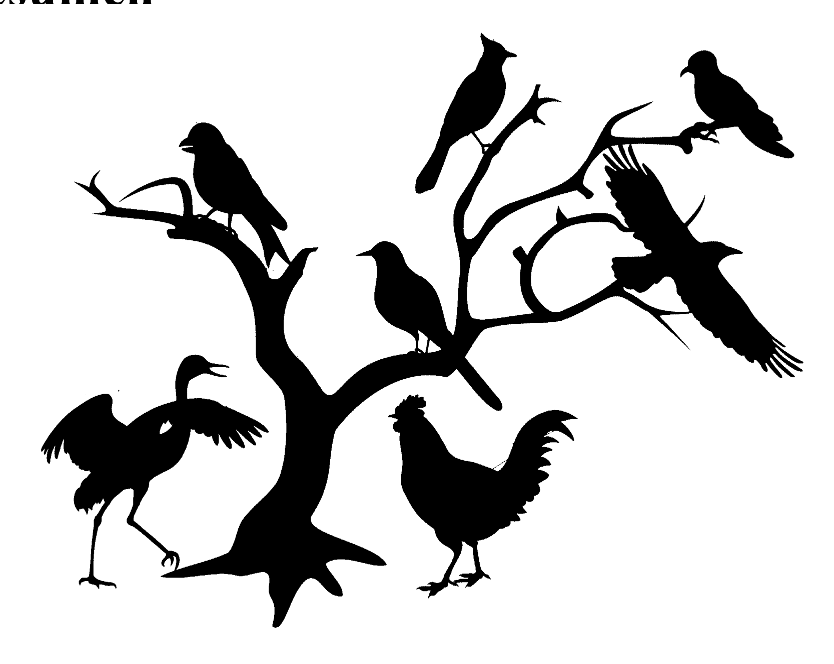
- Siguiendo con el código del apartado anterior se puede afirmar **A)** No se ejecuta el destructor **B)** Si se ejecuta el destructor **C)** Para que se ejecute el destructor debe escribirse obligatoriamente en C++ **D)** El objeto A310 es inmortal y no se destruye nunca **E)** Todas las respuestas anteriores son falsas.

Referencias

- [Booch 94] G.Booch. *Object-oriented analysis and design with applications.* Benjamin Cummings (1994). Versión castellana: *Análisis y diseño orientado a objetos con aplicaciones.* 2ª Edición. Addison-Wesley/ Díaz de Santos (1996).
- [Cueva 93] Cueva Lovelle, J.M. García Fuente Mª P., López Pérez B., Luengo Díez Mª C., Alonso Requejo M. *Introducción a la programación estructurada y orientada a objetos con Pascal*. Distribuido por Ciencia-3 (1993).
- [Joyanes 96] L. Joyanes Aguilar. *Programación orientada a objetos. Conceptos, modelado, diseño y codificación en C++*. McGraw-Hill (1996).
- [Meyer 97] B. Meyer *Object-oriented software construction.* Second Edition. Prentice-Hall (1997). Versión castellana: *Construcción de software orientado a objetos.* Prentice-Hall (1998).
- [Rational 97] UML y herramienta Rational/Rose en *www.rational.com*
- [Rumbaught 91] Rumbaught J., Blaha M., Premerlani W., Wddy F., Lorensen W. *Object-oriented modeling and design.* Prentice-Hall (1991). Versión castellana: *Modelado y diseño orientado a objetos. Metodología OMT*. Prentice-Hall (1996)

Tema 5º
Análisis Orientado a Objetos

- **El proceso de desarrollo de software**
- **Detalle del proceso de desarrollo de software**
- **Análisis Orientado a Objetos**
- **Documentos de análisis**
- **Especificación de requisitos o requerimientos**
- **Diagramas de casos de uso**
- **Escenarios y sub-escenarios**
- **Prototipos**
- **Otras técnicas de análisis orientado a objetos**
- **Resumen**

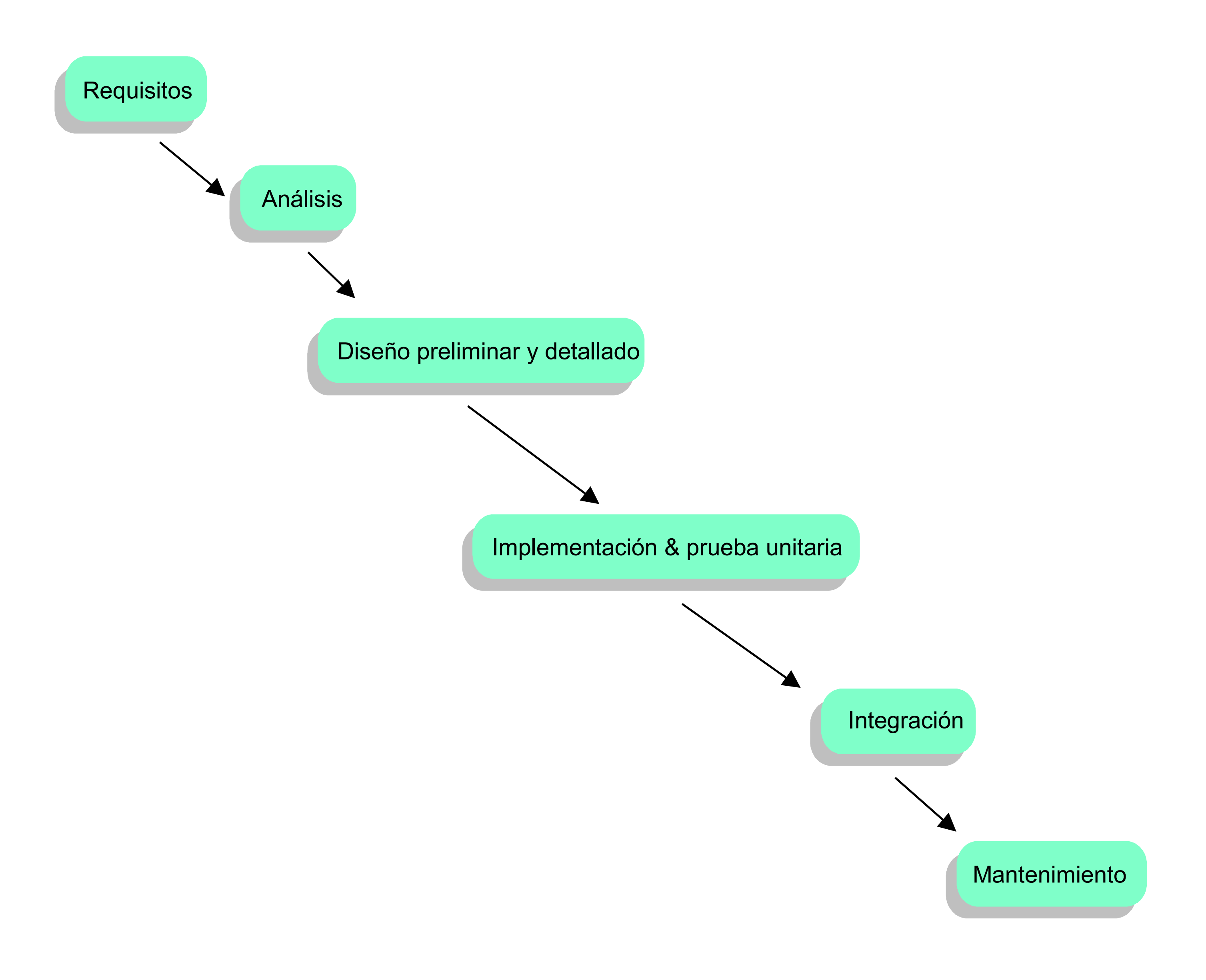


El proceso de desarrollo de software

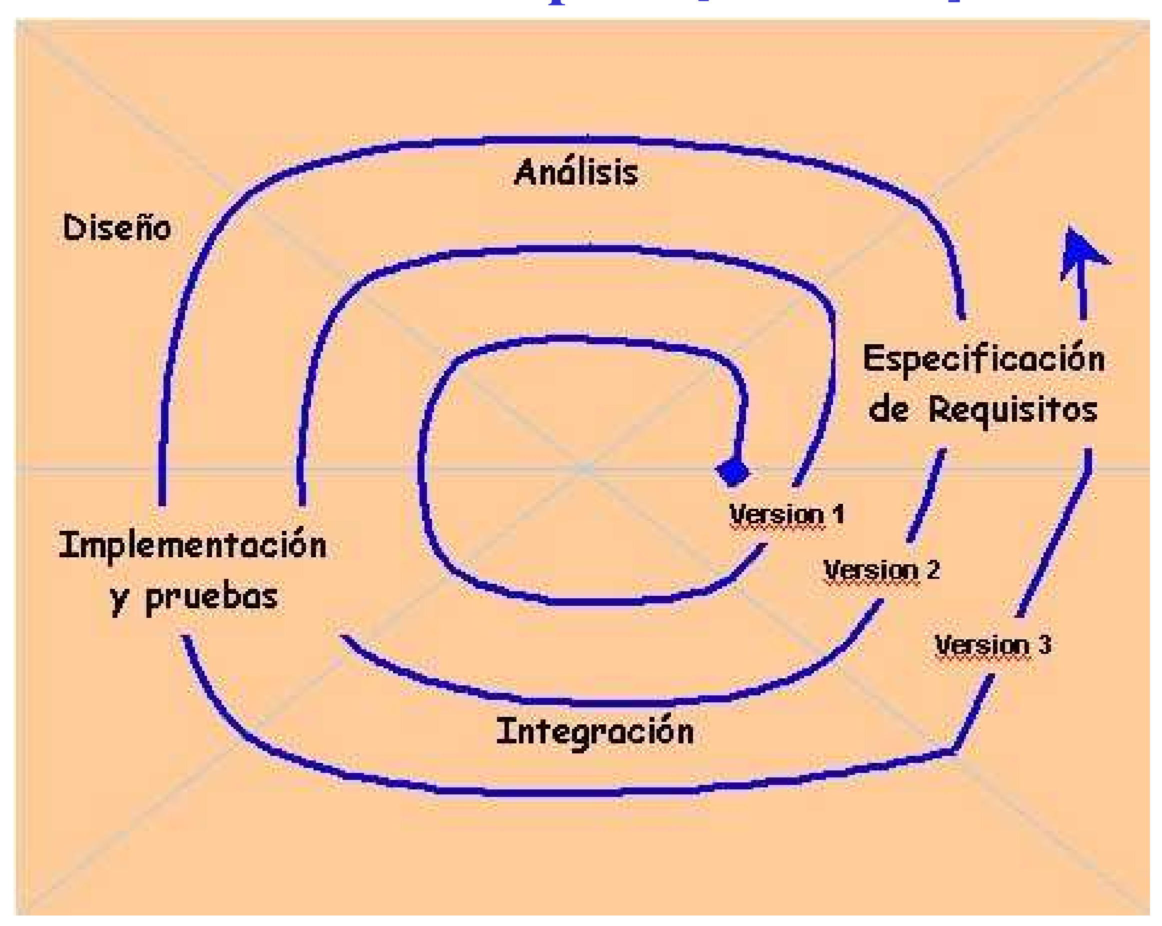
[Booch 94, capítulo 6]

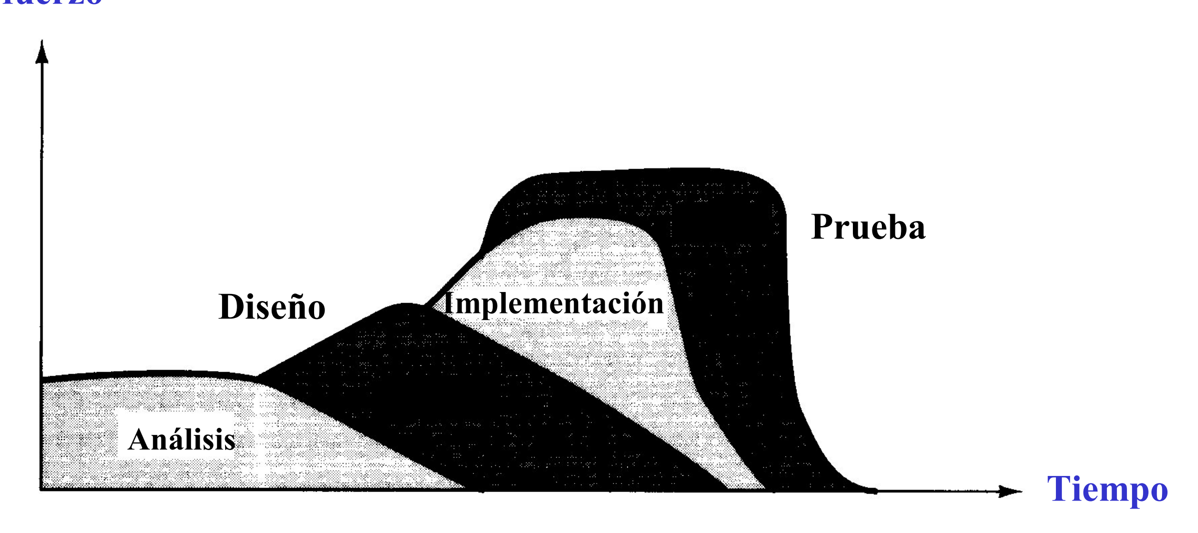
- No hay recetas mágicas, aunque es necesario tener un proceso preceptivo.
- Las características fundamentales de un proyecto con éxito
  - **Buena visión arquitectónica**
    - No existe ningún camino bien definido para idear una arquitectura. Tan sólo se pueden definir los atributos de una buena arquitectura:
      - Capas de abstracción bien definidas
      - Clara separación de intereses entre interfaz e implementación
      - Arquitectura simple
    - Es necesario distinguir entre decisiones estratégicas y tácticas
    - ***Decisiones estratégicas*** *es aquella que tiene amplias implicaciones estratégicas e involucra así a la organización de las estructuras de la arquitectura al nivel más alto*
    - ***Decisiones tácticas*** *son las que sólo tienen implicaciones arquitectónicas locales, es decir sólo involucran a los detalles de interfaz e implementación de una clase*
  - **Ciclo de vida incremental e iterativo**
    - Los ciclos de desarrollo no deben ser anárquicos ni excesivamente rígidos
    - Cada pasada por un ciclo análisis/diseño/evolución lleva a refinar gradualmente las decisiones estratégicas y tácticas, convergiendo en última instancia hacia una solución con los requisitos reales de los usuarios finales (habitualmente no expresados explícitamente por éstos)

El proceso de desarrollo de software
Modelo en cascada



El proceso de desarrollo de software
Modelo en espiral [Jacobson 92]





Detalle de un proceso de desarrollo de software

Aunque el proceso es iterativo el orden de los pasos fundamentales es el siguiente:

- Características comunes de todos los documentos.
  - Identificación. Título, descripción, versión, fecha, revisión, código del documento..
- **Análisis**
  - Documentos de análisis
  - Especificación de requisitos o requerimientos
  - Diagramas de casos de uso
  - Escenarios y sub-escenarios
  - Prototipos
- **Diseño (preliminar y detallado)**
  - **División en módulos y para cada módulo**
    - Modelado de Clases, Objetos y mecanismos de colaboración
      - Diagramas de interacción
        - Diagramas de secuencia
        - Diagramas de colaboración
      - Diagramas de Clases y consulta de patrones de diseño.
      - Diagramas de objetos
    - Modelado del comportamiento de clases y objetos
      - Diagramas de actividades
      - Diagramas de estados
  - **Construcción del modelo físico**
    - Diagramas de componentes
    - Diagramas de despliegue
- **Implementación**
  - Las decisiones iniciales de implementación se toman a partir de los diagramas de componentes y de despliegue
  - Se implementan las clases de un componente a partir de los diagramas de clases y diagramas de objetos
  - A partir de los diagramas de actividades y de los diagramas de estados se implementa el comportamiento de los métodos de cada clase
- **Prueba**
  - Prueba unitaria de cada clase
  - Prueba unitaria de módulos
  - Prueba de integración se realiza siguiendo los escenarios, diagramas de interacción., actividades y estados
- **Mantenimiento**
  - Informes de errores
  - Nueva especificación de requisitos. Nueva versión

Análisis Orientado a Objetos

- **Análisis orientado a objetos (AOO)**
  - *"es un método de análisis que examina los requisitos desde la perspectiva de las clases y objetos que se encuentran en el vocabulario del dominio del problema"* *[Booch 1994]*

- Documentos básicos de análisis orientado a objetos
  - Documentos de análisis
    - Documentación y actas de reuniones
  - Especificación de requisitos o requerimientos
  - Diagramas de casos de uso
  - Escenarios y sub-escenarios
  - Prototipos y su evaluación
- Todos los documentos deben estar identificados y codificados

Identificación

- Es necesario identificar todos los elementos del proceso de desarrollo de software de una forma unívoca
- Todos los documentos deben estar identificados
- Título
  - debe reflejar de la mejor forma posible sus fines y su funcionalidad
- Descripción
- Autores
- Versión. Notación decimal.
- Revisión. Autores
- Fecha
- Código de cada documento o diagrama

Documentos de análisis

- Contiene la documentación que aporta el cliente que encarga la aplicación
- También contiene las actas de las reuniones de trabajo del grupo de análisis
  - Es necesario un secretario que tome acta
  - Es necesario aprobar el acta de cada reunión por todos los miembros

Ejemplo de documento de análisis

Se debe realizar un sistema capaz de mantener una base de datos con todos los equipos hardware y software de una empresa, de manera que se pueda obtener información acerca del número de licencias instaladas y de los equipos en los que están instaladas dichas licencias.

Además debe ser posible controlar el hardware, las modificaciones efectuadas en los equipos, las averías de dichos equipos, la composición de cada uno de los ordenadores y el software que está instalado en ellos.

Por otro lado cada equipo y cada software que posee la empresa tiene asociados una serie de manuales de los que es necesario seguir la pista, pudiendo, en cada momento, saber qué manuales tiene cada equipo y también cada programa.

Por tanto existen tres elementos importantes implicados en el sistema:

1. El software.
2. El hardware.
3. Los manuales.

Es necesario seguirles la pista a estos tres elementos y saber en todo momento las relaciones entre ellos para poder localizar, mediante el ordenador el manual de un componente instalado en un ordenador.

Para el Software es necesario saber el nombre del producto, la versión, la marca o casa que lo fabrica, la fecha de compra, el precio de compra, el proveedor, el soporte (disquetes, CD-ROM, etc.), el número de elementos del soporte, la localización física del soporte, el número de instalaciones, los equipos en los que está instalado, el número de licencias adquiridas, los manuales que acompañan el producto y la localización física de dichos manuales.

Las localizaciones físicas pueden ser sustituidas por los códigos si se codifican tanto los soportes físicos como los manuales.

El sistema debe ser capaz de contestar a las preguntas:

1. Licencias existentes, en uso y necesarias (si procede) de cada una de las aplicaciones que se estén usando en la empresa.
2. Ordenador u ordenadores (si hay varios) en que reside una aplicación.
3. Composición del paquete original (disquetes, CD-ROM, etc. y manuales).
4. Proveedor que sirvió el programa, fecha y precio de adquisición.

Un detalle importante a tener en cuenta es que existe la posibilidad de que exista software llave-en-mano, y en este caso además hay que saber si se dispone o no de los códigos fuente, la casa que lo desarrolló, quién posee los derechos de copia, el precio, el tiempo de desarrollo y el nombre de la persona responsable del proyecto.

Los software se quedan obsoletos y, por tanto, es necesario actualizarlos. Se debe tener en cuenta que es necesario que el sistema ofrezca una reseña histórica del producto (versiones anteriores) y por tanto es necesario saber el estado de cada uno de ellos (activo, actualizado, desechado, en preparación, pedido, etc.)

En el caso de los antivirus y otros programas similares, es necesario obtener regularmente una adaptación, por lo que es importante que el sistema nos avise de la inminencia de la caducidad de dichos sistemas.

Ejemplo de Documento de Análisis (continuación ...)

De cada ordenador se necesita saber su composición (monitor, teclado, ratón y unidad central). De esta última es necesario saber su composición (VGA, disco duro, disquete, placa madre, procesador(es), memoria RAM, memoria caché, etc.).

Cada uno de los cuatro componentes principales estará codificado adecuadamente para permitir el intercambio de dichos equipos entre los diferentes puestos de ordenador, de manera que la asociación no sea fija. De ellos es necesario saber (cuando exista) la marca, el modelo, el número de serie, y otras características particulares (por ejemplo, del monitor la resolución, si es o no *Energy*, etc.)

Además de ordenadores existen otros equipos: impresoras, plotters, scaners y unidades de almacenamiento (ZIP, CD y Magneto-ópticos) de los cuales es necesario saber, al menos, la marca, el modelo, el número de serie y una breve descripción de sus características.

De cada uno de los equipos es necesario tener un reseña histórica de sus averías y cambios, así como una estimación de su precio (en función del precio de compra de cada uno de sus componentes).

En el caso de los ordenadores es necesario saber el software que tienen instalado (comenzando por el sistema operativo) y debe ser posible seguir la pista de los manuales de cada una de sus partes componentes.

De los manuales sólo es necesario controlar su código, su ISBN (si lo posee), su precio (si es aparte del paquete) y su título. Pero es imprescindible poder obtener información del software y hardware que está relacionado con ellos.

Los manuales deben de ser actualizados según se vaya cambiando el software y el hardware .

El programa debe ser capaz de gestionar el sistema desde diferentes puntos de vista: responsable de informática, mantenimiento y usuario.

El responsable de informática es el encargado de comprar todos los componentes (equipos, software y manuales). Además da estos equipos de alta y debe poder apoyarse en el sistema para gestionar los pedidos.

Para esta última labor debe poder anotar en el sistema que ha pedido un componente a un proveedor (por tanto que está pendiente de recibir), confirmando en la recepción este pedido, además tendrá la capacidad de poder anular un pedido o si se da el caso anularlo

Ejemplo de Documento de Análisis (continuación ...)

Además debe poder obtener informes de inventario de los equipos tasados por el precio de compra menos una amortización del 25% anual (que dejaría al equipo sin valor pasados cuatro años) para los procesadores y del 10% anual para el resto de los equipos.

Además debe poder obtener informe de la composición de cada equipo, del estado de disponibilidad de cada uno de ellos y de el estado con respecto a la garantía del equipo.

El responsable de informática es la única persona que puede dar de alta, modificar y dar de baja los equipos.

La baja de un equipo se dará en el momento en que se avise de la avería y si el equipo no tiene arreglo lo daremos de baja permanente.

Además debe poder obtener toda la información que tienen el resto de los usuarios del sistema (responsable de mantenimiento y usuarios), y tendrá acceso a un buzón de sugerencias sobre el sistema.

El responsable de informática se encarga además de reservar un equipo cuando se solicita por cualquier usuario, para ello tiene que obtener los informes de disponibilidad y composición de equipos.

El responsable de mantenimiento debe poder anotar en todo momento las averías de cada equipo (fecha, hora de comienzo, hora de fin, nombre del mecánico y descripción). Debe poder anotar que un equipo no tiene reparación. Además debe poder obtener informes acerca del tiempo de inactividad de los equipos y acceso al buzón de sugerencias.

El usuario debe poder obtener información de los manuales, software y hardware asociado y disponibilidad de un equipo en concreto . Tendrá acceso al buzón de sugerencias.

Se debe tener en cuenta que:

* El sistema trabajará en un entorno multiusuario.
* Las bases de datos serán un modelo estándar.
* Cada equipo estará compuesto por un monitor, un teclado, un ratón, y una unidad central.
*La unidad central estará formada por una serie de componentes que se describirán de manera textual en un campo al efecto.

Especificación de requisitos o requerimientos

- *"La captura de requisitos es el proceso de averiguar, normalmente en circunstancias difíciles, lo que que se debe construir"* [Jacobson 1999, capítulo 6]
- La captura de requisitos es complicada
  – Los usuarios habitualmente no saben expresar exactamente lo que quieren
  – Es difícil tener una visión global del problema a resolver
- La especificación de requisitos es un documento más técnico y elaborado de los documentos de análisis
- Es importante codificar los requisitos para poder seguirlos a lo largo del proceso de desarrollo de software
- Se puede utilizar una especificación jerárquica
  – Están todos codificados por niveles, al igual que las leyes.
  – Se desea que en las actas quede reflejado lo más exactamente posible el problema a resolver, y que en las reuniones de análisis se determine exactamente que requisitos se añaden o se eliminan
  – Los requisitos relacionados se organizan dentro de un mismo nivel
  – Cada nivel 1 se puede hacer corresponder posteriormente con un caso de uso
  – Cada nivel 2 se puede hacer corresponder posteriormente con un escenario
  – Cada nivel 3 se puede hacer corresponder posteriormente con un sub-escenario

Ejemplos de requisitos

R.0 Requisitos generales

R.0.1 Tendremos en cuenta trabajar con fechas que codifiquen el año con cuatro cifras.

R.0.2 Las unidades monetarias deberán poder trabajar con cifras decimales con vistas al inmediato cambio de moneda que se nos aproxima.

R.1 Gestión de clientes

R.1.0 Requisitos generales de los clientes

R.1.0.1 Los clientes pueden ser fijos o eventuales

R.1.0.2 A los clientes se les asigna un número identificativo

R.1.0.3 Los clientes se definen por D.N.I., nombre, dirección, ciudad, teléfono, y departamento.

R.1.1 Añadir clientes

R.1.1.1 Solamente los Usuarios con permiso de Administrador podrán añadir clientes fijos.

R.1.2 Borrado de clientes.

R.1.2.1 Solamente los Usuarios con permiso de Administrador podrán borrar clientes.

R.1.2.2 Para poder borrar clientes es necesario que este no tenga ningún albarán pendiente de facturación y que estos tengan una antigüedad mayor a cinco años.

R.1.2.3 También es necesario que no tenga ninguna máquina reparándose

R.1.3 Modificar clientes.

R.1.3.1 Solo los usuarios con permiso de Administrador pueden modificar los datos de un cliente.

R.1.4 Buscar clientes.

R.1.5 Paso de cliente fijo a cliente eventual y viceversa.

R.1.5.1 Solo los usuarios con permiso de Administrador pueden hacer clientes fijos.

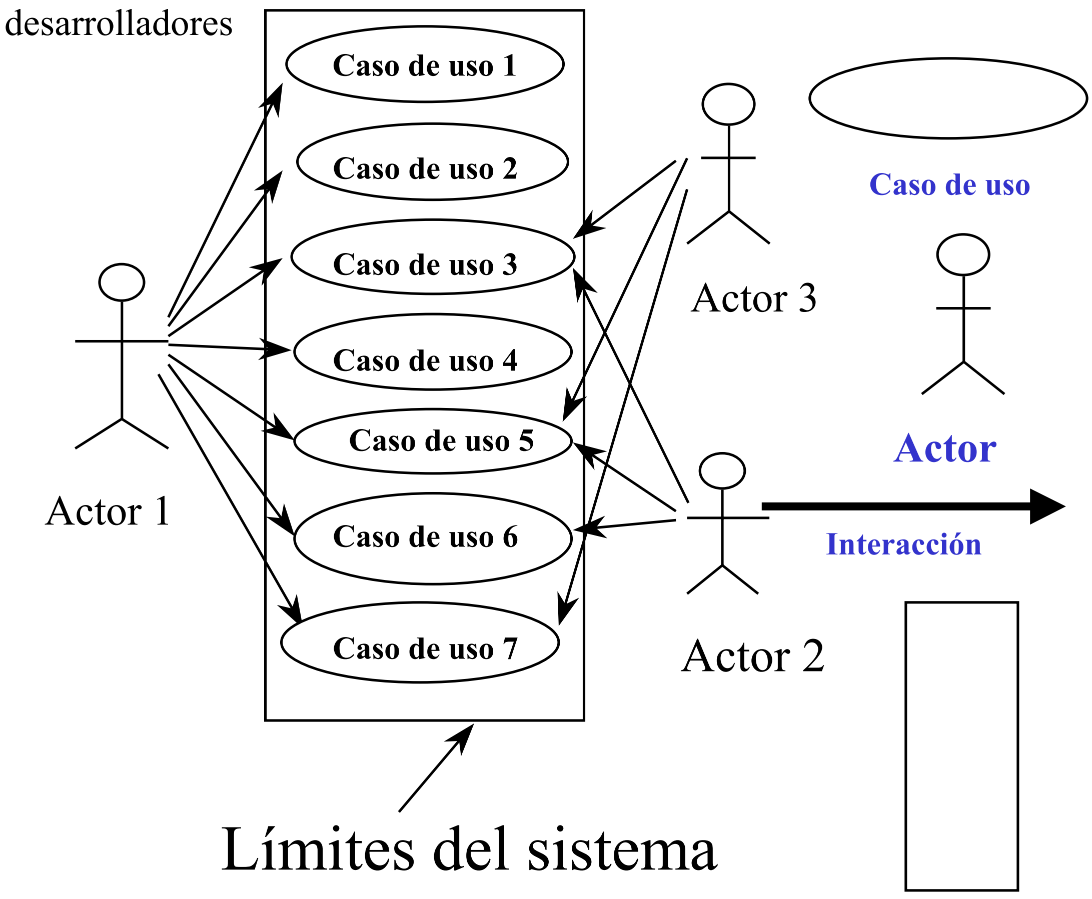
Diagramas de Casos de Uso (I)

[Booch 1999, capítulo 17] [Jacobson 1999, capítulo 3] [Schneider 1998]

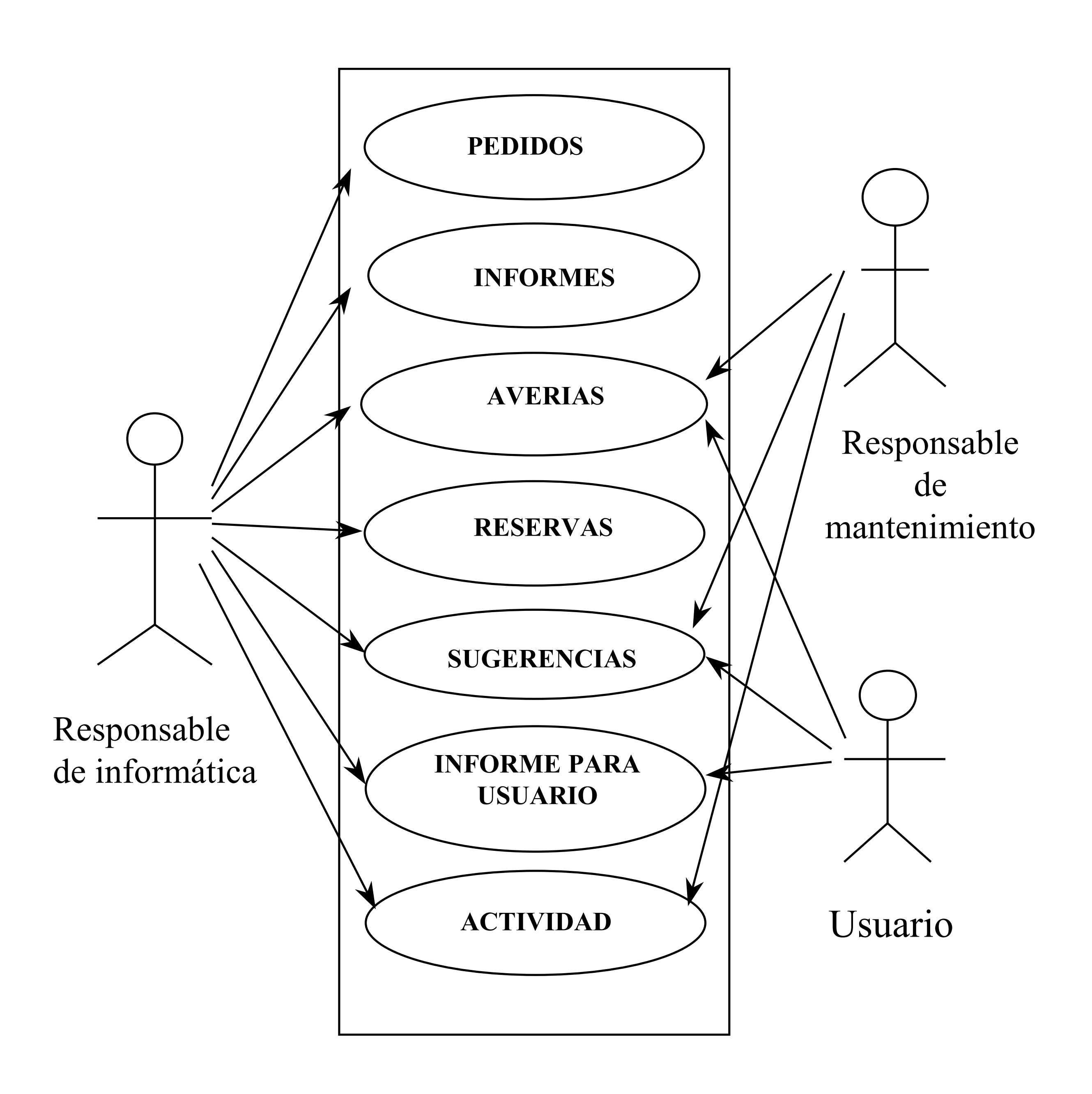
- Es uno de los cinco tipos de diagramas de UML que se utilizan para el modelado de los aspectos dinámicos de un sistema.
- Se corresponden inicialmente con requisitos de primer nivel. Posteriormente se van modelando requisitos de los siguientes niveles.
- Se suelen codificar con el mismo código que el requisito, para hacer más patente su correspondencia.
- Un caso de uso es una técnica de modelado utilizada para describir lo que un nuevo sistema debe hacer o lo que un sistema existente ya hace.
- Los casos de uso representan una vista externa del sistema
- Un modelo de casos de uso se construye mediante un proceso iterativo durante las reuniones entre los desarrolladores del sistema y los clientes (y/o los usuarios finales) conduciendo a una especificación de requisitos sobre la que todos coinciden.
- Un caso de uso captura algunas de las acciones y comportamientos del sistema y de los actores
- El modelado con casos de uso fue desarrollado por Ivar Jacobson [Jacobson 1992]

Diagramas de Casos de Uso (II)

- El sistema que se desea modelar se representa encerrado en un rectángulo
- Los actores son los que interactúan con el sistema. Representan todo lo que necesite intercambiar con el sistema.
  - Un actor es una clase
- Se diferenciará entre actores y usuarios.
  - Un usuario es una persona que utiliza el sistema
  - Un actor representa el papel (rol) que una persona desempeña
  - Por ejemplo una persona puede ser usuario y administrador en un sistema, unas veces actuará como usuario y otras como administrador, pero deben contemplarse ambos actores.
- Los Casos de Uso es un camino específico para utilizar el sistema
- Para cada Caso de Uso, Actor y Sistema se realiza una descripción detallada
- Los Casos de Uso tan sólo indican opciones generales
- El diagrama de Casos de Uso es un diagrama sencillo que tiene como finalidad dar una visión global de toda la aplicación de forma que se pueda entender de una forma rápida y gráfica tanto por usuarios como por desarrolladores



Ejemplo de Casos de Uso



Ejemplo de descripción de los Casos de Uso

1.PEDIDOS

Escenario general donde se realizan todas las operaciones relativas a pedidos: hacer, recibir, anular y devolver pedidos. Todo es realizado por el Responsable de Informática.

2.INFORMES

Todos los informes que son necesarios para el funcionamiento de la empresa: informe de pedido, de amortizaciones, de inactividad, de composición de equipos básicos, de composición de otros equipos, de inventario software y manuales, de garantías y de disponibilidad. Estos informes son realizados para el Responsable de Informática.

3.AVERIAS

Engloba todas ls operaciones relativas a las averías tanto el aviso que puede ser realizado por cualquier actor (Responsable de Informática, de Mantenimiento o Usuario) , como el parte de avería que es realizado por el Responsable de Mantenimiento y entregado al Responsable de Informática.

4.RESERVAS

Es tanto la petición de reserva de un equipo con unas características determinadas, que puede ser realizada por cualquier usuario al Responsable de Informática, como la concesión de una reserva que realiza este último a un usuario.

5.SUGERENCIAS

Es una línea de comunicación entre los diferente agentes que interaccionan con el sistema.

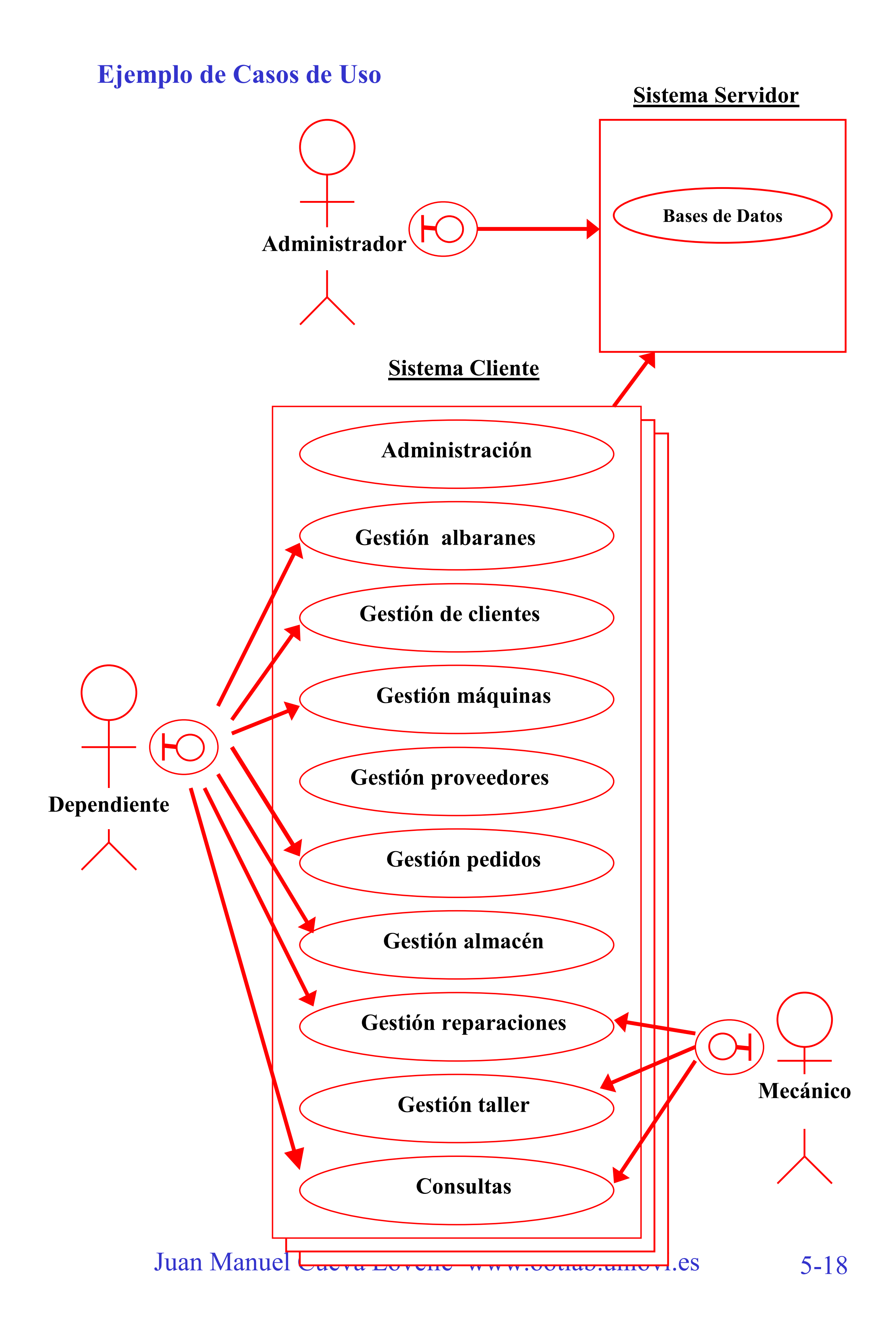
6.INFORMES PARA EL USUARIO

Es un informe especialmente realizado para el usuario donde este puede encontrar toda la información que pueda necesitar en un momento determinado sobre un equipo, su disponibilidad, software o un manual.

7.ACTIVIDAD

Realizado por el Responsable de Informática engloba todo lo relativo al buen funcionamiento del material de la empresa: dar de baja temporalmente un equipo cuando esta en reparación, dar de baja permanentemente un equipo cuando no tiene arreglo y actualizar tanto software como los manuales.

Ejemplo de Casos de Uso



Descripción de actores

Nombre de Actor: Administrador

Definición: Es el encargado de administrar el sistema. Tendrá todos los permisos y libertad de movimientos por el sistema.

Notas:

- El administrador es el encargado de manipular la información contenida en el sistema.
- Tiene acceso a toda la información del sistema y es el único que puede modificar todo lo que le de la gana..

Nombre de Actor: Mecánico

Definición: Es el encargado de realizar las oportunas reparaciones en las máquinas y a su criterio y valoración queda el tomar las decisiones oportunas respecto a que reparación y si es necesario o no el ingreso de la máquina en el taller.

Notas:

- No lo encajamos en la figura de una persona concreta sino que pueden ser varias personas las que puedan encargarse de esta tarea.
- El mecánico solo tiene acceso a la parte del sistema referente a las máquinas a las reparaciones y al las consultas, el acceso al resto le está vedado.
- Dentro de la parte del sistema al que puede acceder no se le permite el borrado de información, solo, añadir y modificar.

Nombre de Actor: Dependiente

Definición: Es la persona que está en contacto directo con los clientes. Tiene acceso limitado a las operaciones del sistema.

Notas:

- El dependiente no podrá dar de alta a clientes fijos, pero si a clientes eventuales.
- No podrá borrar clientes, ni máquinas, ni artículos, ni reparaciones.
- No tiene acceso a la gestión de proveedores ni del taller

Sistemas

SISTEMA SERVIDOR

El Sistema Servidor, (en nuestro caso particular) estará formado por una máquina Windows NT Server (también podría ser una máquina Unix) que será atacado por sistemas Clientes constituidos por máquinas Windows 95 ó Windows NT.

El gestor de la base de datos (CTSQL de MultiBase...), junto a la propia base de datos, se encuentran en el servidor UNIX o Windows NT (solo para el CTSQL).

Existirá también la posibilidad de configurarlo en una sola máquina en Windows 95 ó Windows NT, en cuyo caso los programas de la aplicación, el gestor de la base de datos y la base de datos residirían en la misma máquina Windows.

SISTEMAS CLIENTES

Los programas de la aplicación residen en la máquina «cliente» Windows95 o Windows NT que atacan al servidor donde se encuentra instalada el gestor de la base de datos junto con la base de datos correspondiente.

INTERFACES

INTERFAZ ADMINISTRADOR

El interfaz del Administrador le permite acceder a todas las opciones que presenta la aplicación

INTERFAZ DEPENDIENTE

El Dependiente solamente tendrá acceso a algunas de las funciones que soporta la aplicación y dentro de estas su capacidad de maniobra estará limitada

INTERFAZ MECÁNICO

El mecánico tendrá acceso solamente a las reparaciones y a la gestión del taller además de las consultas.

Ejemplo de descripción de casos de uso

Nombre del Caso de Uso 1: Gestión Clientes

Definición: Actualizaciones de la información acerca de los clientes de Comercial Quirós

Notas:

- Los clientes vienen definidos por: número_cliente, DNI, nombre, dirección, ciudad, teléfono y el departamento para el que trabajan.
- En el caso de que no pertenezca a un departamento determinado este aparecerá en blanco.
- Los clientes pueden ser fijos o eventuales.

Nombre del Caso de Uso 2: Gestión Máquinas

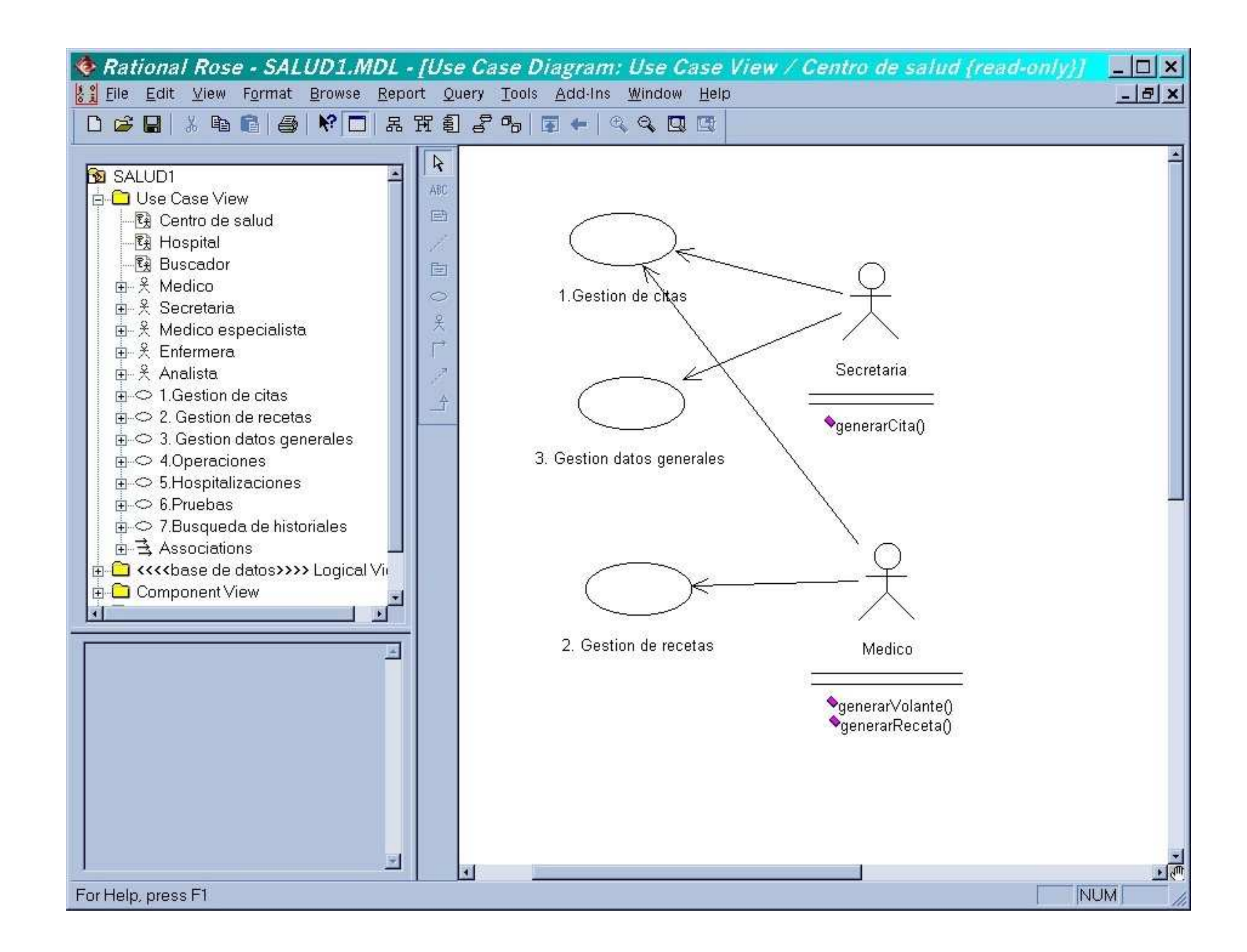
Definición: Actualizaciones de la información acerca de las máquinas de los clientes de Comercial Quirós, y de las cuales nos encargamos de su mantenimiento y reparación.

Notas:

- Las máquinas vienen definidas por: *nº identificatorio, número_cliente, tipo, marca y modelo*.
- Con *número_cliente* relacionamos la máquina con el cliente al que pertenece.
- Un mismo cliente puede tener varias máquinas.
- El *tipo* nos dice a que clase de máquina pertenece (fotocopiadora, fax...)

Casos de uso en Rational Rose ®

• Tiene una sección para ir introduciendo los Casos de Uso (*Use Case View*)
• Permite el manejo de actores, que se traducirán al sistema como clases
• Cada sistema recibe un nombre (no aparece el rectángulo) y está ligado a una ventana



Escenarios y sub-escenarios

- Cada caso de uso da lugar múltiples escenarios
- Se codifican siguiendo la codificación de los casos de uso
- Se estudia cada escenario utilizando guiones como los que se usan en el cine
- Cada equipo que pasa por un escenario identifica los objetos y sus responsabilidades, así como los mecanismos que relacionan los objetos
- De los escenarios iniciales se puede pasar a otros escenarios secundarios
- Los escenarios también se pueden utilizar para probar el sistema en la fase de pruebas
- El estudio de los escenarios con detalle permitirá enriquecer el Diccionario de clases
- No es necesario hacer todos los escenarios y sub-escenarios posibles si se observa que no enriquecen el diccionario de clases

Numeración: 1.1.
Titulo: Hacer pedido
Precondiciones: Sugerencias de compra, caducidad de licencias, bajas permanente hardware, …
Quien Lo Comienza: Responsable de Informática.
Quien Lo Finaliza: Responsable de Informática.
Postcondiciones:
Excepciones:
Descripción: Son las operaciones de compra de todos los componentes (hardware, software y manuales) que realiza el responsable de informática. Además da estos equipos de alta y debe apoyarse en el sistema para gestionar los pedidos correctamente para lo que debe anotar en el sistema que ha pedido un componente a un proveedor (por tanto que está pendiente de recibir).

Numeración: 1.2.
Titulo: Anular pedido
Precondiciones: Cambio de precio, cambio de necesidades de la Empresa.
Quien Lo Comienza: Responsable de Informática
Quien Lo Finaliza: Responsable de Informática
Postcondiciones:
Excepciones
Descripción: Esta operación la realiza el responsable de informática cuando toma la decisión de anular un pedido que había realizado con anterioridad.
de informática confirma la recepción de los pedidos.

Ejemplo de escenarios

Caso de uso 1: Gestión de Clientes

Nombre de Escenario 1.1: Dar de alta un cliente eventual

Precondiciones:
- No existe ficha de cliente.

Postcondiciones:
- Todos los datos se han introducido correctamente.
- El numero de clientes se incrementa en uno

Excepciones:

Iniciado por: Dependiente/Administrador.

Finalizado por: Dependiente/Administrador.

Detalle operaciones:
- Cliente acude a una tienda de la compañía.
- Dependiente (ó Administrador) obtiene datos de cliente.
- Dependiente (ó Administrador) introduce ficha en el sistema con los datos *número, dni, nombre, dirección, ciudad, teléfono*, y *departamento.*

Nombre de Escenario 1.2: Dar de alta un cliente fijo.

Precondiciones:
- No existe ficha de cliente.

Postcondiciones:
- Todos los datos se han introducido correctamente.
- El número de clientes se incrementa en 1

Excepciones

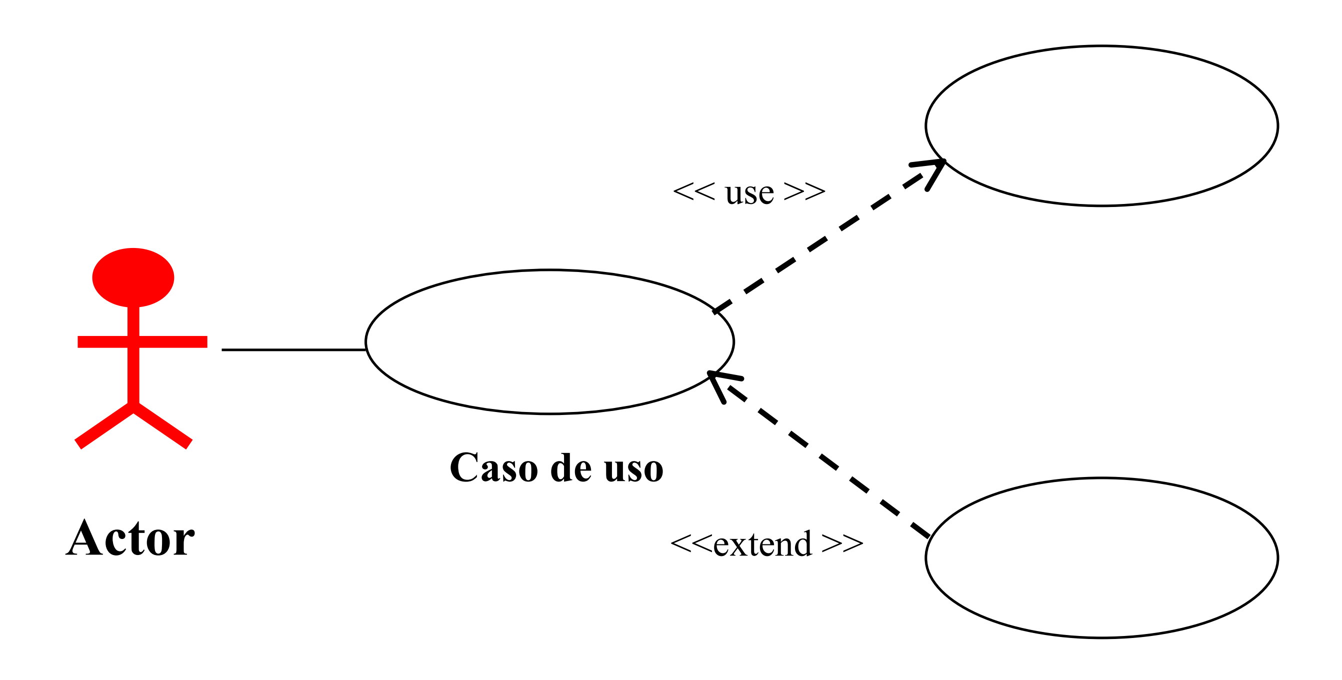
Iniciado por: Administrador.

Finalizado por: Administrador.

Detalle operaciones:
- Cliente acude a una tienda de la compañía.
- Administrador obtiene los datos del cliente.
- Administrador introduce ficha en el sistema con los datos *número, dni, nombre, dirección, ciudad, teléfono*, y *departamento.*

Diagramas de Casos de Uso (III)

- Un **caso de uso** es la típica interacción entre un usuario y un sistema informático
- Un **actor** es el papel que el usuario juega con respecto al sistema. Un actor no tiene que ser un humano, puede ser por ejemplo otro sistema externo que pide información al sistema actual
- La relación **<<*extend*>>** se utiliza cuando un caso de uso es similar a otro caso de uso pero se le añade alguna característica nueva
- La relación **<< *use* >>** se utiliza cuando se tiene una parte del comportamiento común a más de un caso de uso, y no se desea almacenar una copia en cada caso de uso de la descripción de este comportamiento.



Prototipos

[Piattini 96]

- El prototipado consiste en la elaboración de un modelo o maqueta del sistema que se construye para evaluar mejor los requisitos que se desea que cumpla
- Es particularmente útil cuando:
  - El área de la aplicación no está bien definida, bien por su dificultad o bien por falta de tradición en su automatización.
  - El coste del rechazo de la aplicación por los usuarios, por no cumplir sus expectativas, es muy alto.
  - Es necesario evaluar previamente el impacto del sistema en los usuarios y en la organización.
- Estos modelos o prototipos suelen consistir en versiones reducidas, demos o conjuntos de pantallas (que no son totalmente operativos) de la aplicación pedida. Existen tres razones principales para emplear prototipado, ordenadas por frecuencia de uso:
  - **Prototipado de la interfaz de usuario** para asegurarse de que esta bien diseñada, que satisface las necesidades de quienes deben usarlo. Este tipo de prototipado es bastante frecuente, no cuesta mucho y puede consistir en simples modelos de pantallas en papel, simuladas con programas de dibujo o presentación o auténticas simulaciones muy elaboradas de la interfaz. No suele resultar más caro que el trabajo tradicional y es muy efectivo para evitar los múltiples cambios que suelen solicitar los usuarios en este aspecto.
  - **Modelos de rendimiento** para evaluar el posible rendimiento de un diseño técnico, especialmente en aplicaciones críticas en este aspecto. Estos modelos tienen un carácter puramente técnico y, por lo tanto, no son aplicables al trabajo de análisis de requisitos.
  - **Prototipado funcional**. Cada vez más utilizado, está relacionado con un ciclo de vida iterativo. En este caso, en vez de seguir el procedimiento habitual (tirar el prototipo una vez probado y empezar a desarrollar la aplicación), el prototipo supone una primera versión del sistema con funcionalidad limitada. A medida que se comprueba si las funciones implementadas son las apropiadas, se corrigen, refinan o se añaden otras nuevas hasta llegar al sistema fina



Otras técnicas de análisis orientado a objetos

- **Análisis OO mediante fichas CRC (Clases/Responsabilidades/Colaboradores)**
- **Descripción informal en lenguaje natural**

Análisis mediante fichas CRC (Clases/Responsabilidades/Colaboradores)

[Wirfs-Brock 1990] [Wilkinson 1995, capítulo 4] [Bellin 1997]

- Es una forma simple de analizar escenarios
- Son muy útiles para la enseñanza del AOO, DOO y POO
- Facilitan las "tormentas de ideas" y la comunicación entre desarrolladores
- Se crea una ficha para cada clase que se identifique como relevante en el escenario
- A medida que el equipo avanza puede dividir las responsabilidades
- Las fichas CRC pueden disponerse espacialmente para representar relaciones de colaboración
- Las fichas CRC también se pueden colocar para expresar jerarquías de generalización/especialización
- No están contempladas en UML

Ficha CRC (anverso y reverso)

Clase
Responsabilidades
Colaboraciones

Clase
Superclase Subclase
Atributos

Ejemplo de ficha CRC

Clase: Reunión
Responsabilidades Planificar Comprobar la sala asignada Conocer hora de comienzo Conocer la fecha Conocer número de asistentes Conocer equipamiento necesario
Colaboraciones Sala de conferencias Organizador de reuniones

Clase: Reunión
Superclase: Subclases: Reunión de trabajo, Junta de Escuela, Clase de un curso
Atributos: Orden del día Lugar Fecha Hora de inicio Asistentes Equipo necesario

Descripción informal en lenguaje natural

[Abbott 1983]

- Subrayar los nombres y los verbos de la descripción del problema
- Los **nombres** representan los **objetos** candidatos
- Los **verbos** representan las **operaciones** candidatas
- Ventajas
  - Obliga al desarrollador a trabajar sobre el vocabulario del espacio del problema
  - Es sencillo
  - Es didáctico
- Inconvenientes
  - No es riguroso, al ser el lenguaje natural ambiguo
  - Es compleja su aplicación a grandes proyectos

Resumen

- No existen recetas fáciles para el análisis de software
- La correcta definición de los requisitos y su seguimiento en el proceso de desarrollo es uno de los factores fundamentales de la calidad del software
- Los escenarios son una potente herramienta para el análisis orientado a objetos, y pueden utilizarse para guiar los procesos de análisis clásico, análisis del comportamiento y análisis de dominios.
- Las abstracciones clave reflejan el vocabulario del dominio del problema y pueden ser descubiertas en el dominio del problema, o bien ser inventadas como parte del diseño.
- Los mecanismos denotan decisiones estratégicas de diseño respecto a la actividad de colaboración entre muchos tipos diferentes de objetos
- El único artefacto que tiene UML a nivel de análisis son los Diagramas de Casos de Uso.

EJERCICIOS PROPUESTOS

5.1 Realizar el análisis del juego del ajedrez. Se puede jugar dos personas entre sí o una persona contra el ordenador. En este último caso debe ser posible seleccionar el nivel de dificultad entre una lista de varios niveles. El juego de ajedrez permitirá al jugador elegir el color de las piezas. La aplicación deberá permitir detectar los movimientos ilegales de las piezas, tiempos utilizados cada jugador, registro de jugadas y piezas perdidas. También determinará si se alcanzan tablas, y permitirá abandonar la partida a un jugador.

5.2 Realizar el análisis de una aplicación que realiza estudios de mercado para situar grandes superficies (hipermercados). Se supone que cada gran superficie necesita un mínimo de población que pueda acceder a dicho hipermercado en menos de un tiempo dado. La red de carreteras se puede representar mediante un grafo.

5.3 Realizar el análisis para gestionar los fondos bibliográficos y de socios de una biblioteca por Internet.

5.4 Realizar el análisis para gestionar un centro de enseñanza (profesores, asignaturas, alumnos, matriculación, calificaciones de cada asignatura, expediente,...) por Internet.

5.5 Realizar el análisis del juego de cartas del tute, para poder jugar con el ordenador.

Referencias Bibliográficas

- [Abbott 1983] Abbott, R.J.Program Design by Informal English Descriptions. *Comunications of the ACM, 26(11),882-894. 1983.*
- [Bock/Odell 1994] C. Bock and J. Odell, "A Foundation For Composition," *Journal of Object-oriented Programming*, October 1994.
- [Booch 1994] G.Booch. *Object-oriented analysis and design with applications.* Benjamin Cummings (1994). Versión castellana: *Análisis y diseño orientado a objetos con aplicaciones.* 2ª Edición. Addison-Wesley/ Díaz de Santos (1996).
- [Booch 1999] G. Booch, J. Rumbaugh, I. Jacobson. *The unified modeling language user guide.* Addison-Wesley (1999). Versión castellana *El lenguaje unificado de modelado.* Addison-Wesley (1999)
- [Bellin 1997] D. Bellin and S. Suchman Simone.*The CRC Card book*. Addison-Wesley, 1997
- [Coad 1991] P. Coad, E. Yourdon. *Object-Oriented Analysis*. Second Edition .Prentice-Hall, 1991.
- [Cook 1994] S. Cook and J. Daniels, *Designing Object Systems: Object-oriented Modelling with Syntropy,* Prentice-Hall Object-Oriented Series, 1994.
- [Eriksson 1998] H-E Eriksson & M. Penker. *UML Toolkit. Wiley, 1998.*
- [Fowler 1997] M. Fowler with K. Scott, *UML Distilled: Applying the Standard Object Modeling Language*, ISBN 0-201-32563-2, Addison-Wesely, 1997. Versión castellana *UML gota a gota,* Addison-Wesley 1999.
- [Jacobson 1992] I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard. *Object-Oriented Software Enginneering. A use Case Driven Approach.*, ISBN 0-201-54435-0, Addison-Wesely, 1992.
- [Jacobson 1999] I. Jacobson, G. Booch, J. Rumbaugh. *The unified software development process*. Addison-Wesley (1999).Versión Castellana. *El Proceso Unificado de Desarrollo de Software*. Prentice-Hall, 2000.
- [Lee 1997] R. C.Lee & W. M. Tepfenhart. *UML and C++, Prentice-Hall, 1997*
- [Piattini 1996] M.G. Piattini, J.A. Calvo-Manzano, J. Cervera, L. Fernández. *Análisis y diseño detallado de aplicaciones de gestión*. RA-MA (1996)
- [Rumbaught 1991] Rumbaught J., Blaha M., Premerlani W., Wddy F., Lorensen W. *Object-oriented modeling and design.* Prentice-Hall (1991). Versión castellana: *Modelado y diseño orientado a objetos. Metodología OMT*. Prentice-Hall (1996)
- [Rumbaught 1999] Rumbaught J., I. Jacobson, G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley (1999). Versión castellana. *El Lenguaje Unificado de Modelado. Manual de Referencia.*Addison-Wesley, 2000.
- [Reenskaug 1996] T. Reenskaug. *Working with Objects. The Ooram Software Engineering Method.*Prentice-Hall, 1996
- [Schneider 1998] G. Schneider, J. Winters. *Applying Use Cases: A Practical Approach*. Addison-Wesley, 1998.
- [Wilkinson 1995] N. M. Wilkinson. *Using CRC Cards. An Informal Approach to Object-Oriented Development*, 1995, SIGS BOOKS, ISBN 1-884842-07-0
- [Wirfs-Brock 1990] R. Wirfs-Brock, B. Wilkerson y L. Wiener. *Designing Object-Oriented Software.*Pentice-Hall, 1990.

Referencias en la web

- [Coad] Peter Coad http://www.togethersoft.com/
- [OMG] Object Management Group, http://www.omg.org
- [ROSE] Herramienta Rational Rose http://www.rational.com
- [Shlaer-Mellor] Shlaer-Mellor Object-Oriented Analysis http://www.projtech.com
- [UML] UML en www.rational.com y en http://www.omg.org

Tema 6º: Diseño Orientado a Objetos

- Diseño preliminar y Diseño detallado
- Modelado de la Arquitectura del Sistema
- Abstracciones y mecanismos clave
- Elementos básicos del Diseño Orientado a Objetos
  - Diagramas de interacción
    - Diagramas de secuencia
    - Diagramas de colaboración
  - Diagramas de Clases y consulta de patrones de diseño.
  - Diagramas de objetos
  - Diagramas de actividades
  - Diagramas de estados
  - Diagramas de componentes
  - Diagramas de despliegue

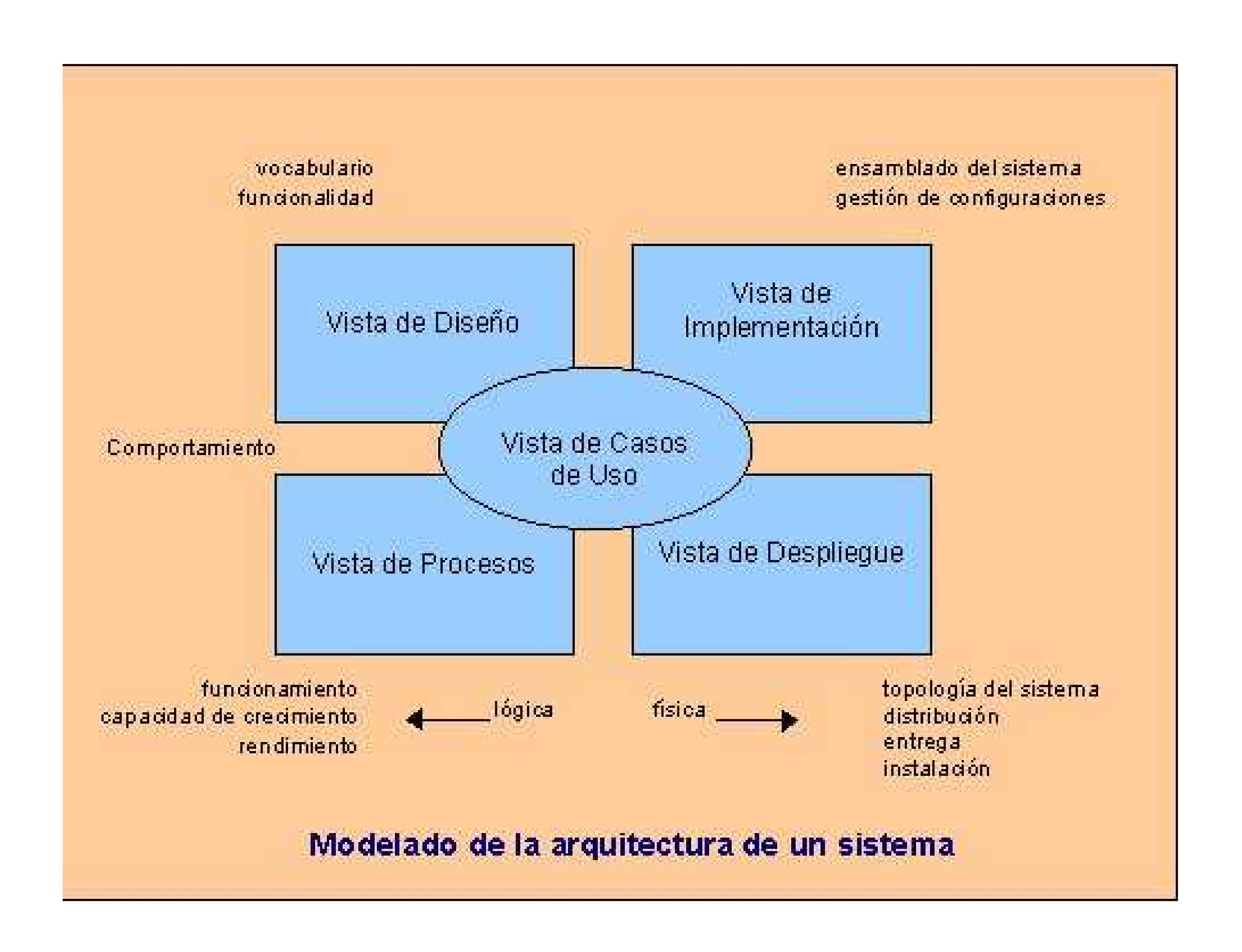


Diseño Orientado a Objetos

- Diseño preliminar
  - Incluye los mismos diagramas del diseño detallado pero a nivel más sencillo (con menos detalle)
  - Es necesario para estimar costes y tiempos antes de realizar el diseño detallado
  - Es el primer paso dentro del proceso iterativo de diseño
- Diseño detallado
  - Es un refinamiento sucesivo de diagramas de diseño

Modelado de la Arquitectura del Sistema

[Booch 99, capítulos 1,2, 31]



Modelado de la Arquitectura del Sistema

Cinco vistas interrelacionadas

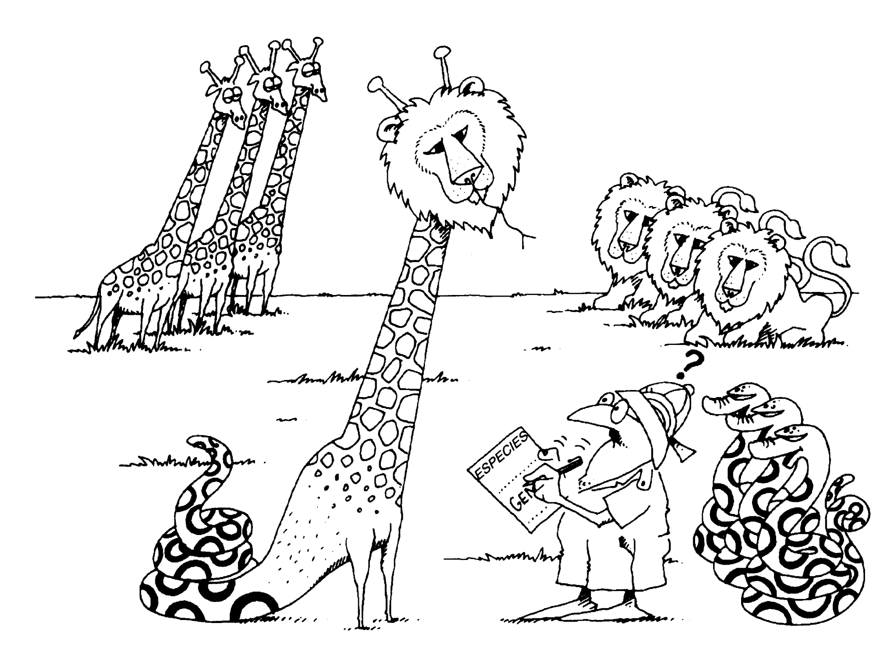
[Booch 99, capítulos 1,2, 31]

- **Vista de casos de uso**
  - Muestra los requisitos del sistema tal como es percibido por los usuarios finales, analistas, y encargados de pruebas. Utiliza los diagramas:
    - Caos de uso
    - Diagramas de Interacción
    - Diagramas de estados
    - Diagramas de actividades
- **Vista de diseño**
  - Captura el vocabulario del espacio del problema y del espacio de la solución. Utiliza los diagramas
    - Diagramas de clases
    - Diagramas de objetos
    - Diagramas de interacción
    - Diagramas de estados
    - Diagramas de actividades
- **Vista de procesos**
  - Modela la distribución de los procesos e hilos (*threads*)
  - Emplea los mismos diagramas de las vista de diseño, pero poniendo especial atención en las clases activas y los objetos que representan hilos y procesos
- **Vista de implementación**
  - Modela los componentes y archivos que se utilizan para ensamblar y hacer disponible el sistema físico. Emplea los diagramas:
    - Para modelar aspectos estáticos: Diagramas de Componentes
    - Para modelar aspectos dinámicos:
      - Diagramas de Interacción
      - Diagramas de Estados
      - Diagramas de Actividades
- **Vista de despliegue**
  - Modela los nodos de la topología hardware sobre la que se ejecuta el sistema
    - Para modelar aspectos estáticos: Diagramas de Despliegue
    - Para modelar aspectos dinámicos:
      - Diagramas de Interacción
      - Diagramas de Estados
      - Diagramas de Actividades

La importancia de una clasificación correcta (I)

[Booch 1994]

- La identificación de clases y objetos es la parte más difícil del diseño orientado a objetos
- La identificación de clases y objetos implica descubrimiento e invención
- No hay recetas fáciles para identificar clases y objetos
- Clasificar es agrupar cosas que tienen una estructura común o exhiben un comportamiento común
- La clasificación ayuda a identificar jerarquías de generalización, especialización y agregación entre clases
- La clasificación también proporciona una guía para tomar decisiones sobre modularización

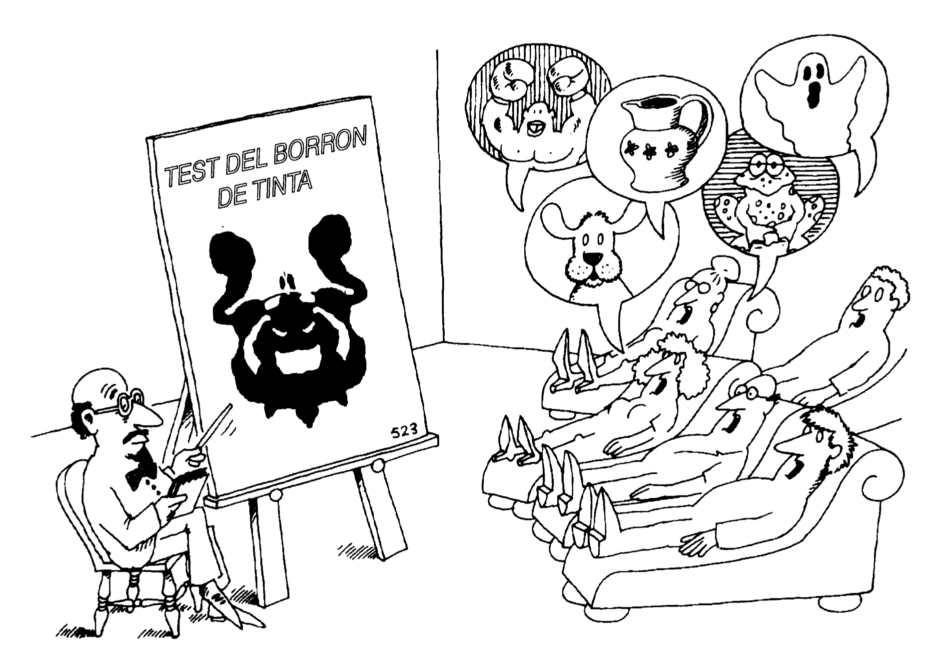


La clasificación es el medio por el cual ordenamos el conocimiento.

La importancia de una clasificación correcta (II)

[Booch 94]

- La dificultad de la clasificación
  - Un objeto debe tener una frontera o interfaz nítida
  - Sin embargo a veces las fronteras son difusas
  - El descubrimiento de un orden no es tarea fácil…; sin embargo, una vez que se ha descubierto el orden, no hay dificultad alguna en comprenderlo
- Ejemplos de clasificación
  - Clasificación de los organismos en géneros y especies
- La naturaleza incremental e iterativa de la clasificación
  - Cualquier clasificación es relativa a la perspectiva del observador que la realiza
  - No hay clasificaciones perfectas (unas son mejores que otras en función del dominio del problema)

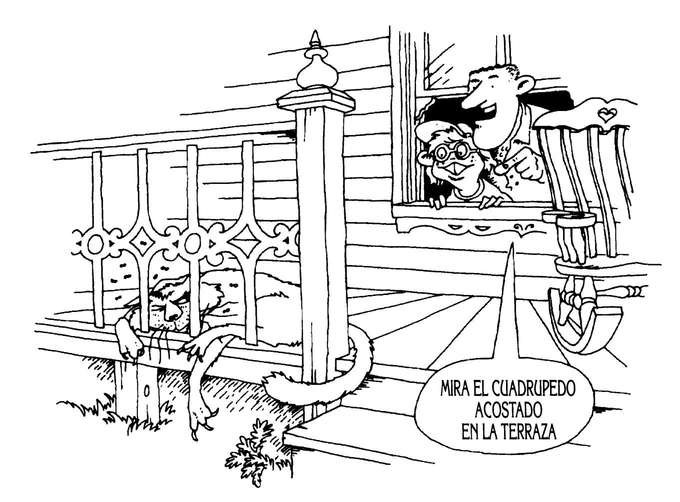


Diferentes observadores pueden clasificar el mismo objeto de distintas formas.

Abstracciones clave

son clases u objetos que forman parte del dominio del problema [Booch 94]

- Identificación de las abstracciones clave
  - Búsqueda de las abstracciones clave
    - Depende de las características de cada dominio
    - Descubrir las abstracciones de los expertos en el dominio
    - Se inventan nuevas abstracciones que son útiles para el diseño o la implementación
    - Se compara con otras abstracciones buscando similitudes
    - Se recomienda el uso de escenarios para guiar el proceso de identificación de clases y objetos
  - Refinamiento de las abstracciones clave
    - Cada abstracción debe ubicarse en el contexto de la jerarquía de clases y objetos
    - La colocación de clases y objetos en los niveles correctos de abstracción es una tarea difícil
    - Se recomienda elegir los identificadores con el siguiente criterio
      - **objetos:** *los identificadores deben ser **nombres propios***
      - **clases:** *los identificadores deben ser **nombres comunes***
      - **operaciones de modificación:** *los identificadores deben ser **verbos activos***
      - **operaciones de selección:** *los identificadores deben ser **verbos interrogativos o verbos en forma pasiva***

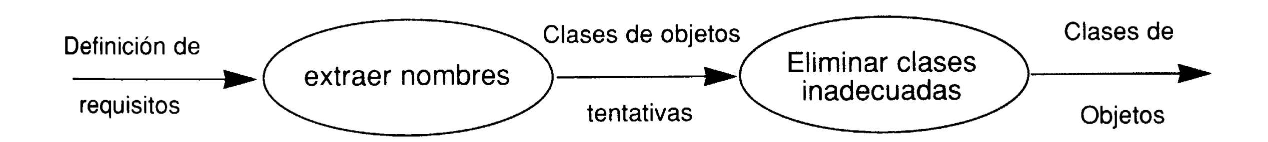


Las clases y objetos deberían estar al nivel de abstracción adecuado: ni demasiado alto ni demasiado bajo.

Identificación de clases y objetos

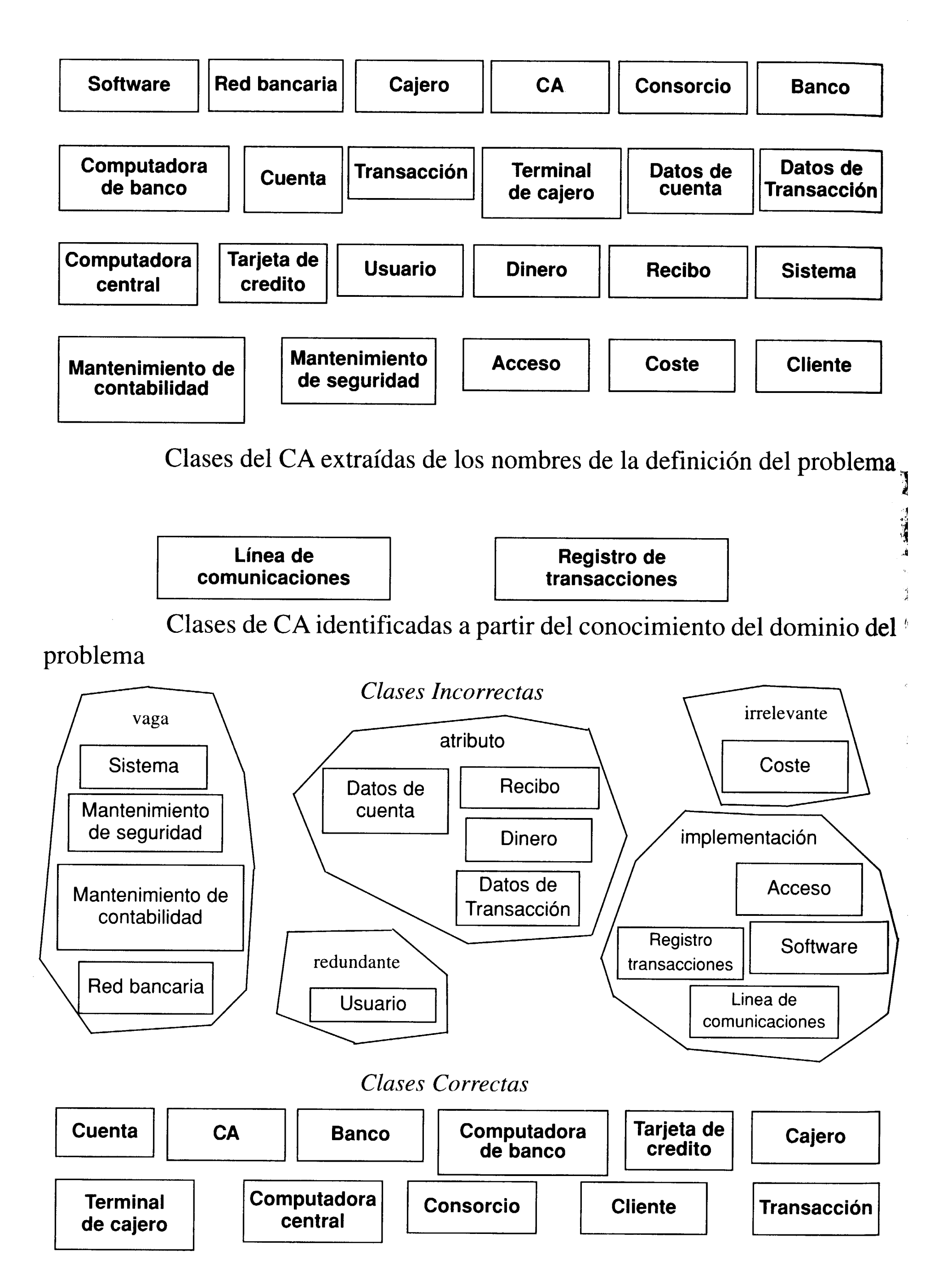
[Rumbaught 91]

- Enumerar los candidatos a clases (siguiendo uno de los métodos de análisis, por ejemplo subrayando sustantivos)
- No se preocupe demasiado por la herencia, ni por las clases de alto nivel
- Eliminar clases redundantes
- Eliminar clases irrelevantes
- Eliminar clases vagas
- Identificar atributos



Ejemplo cajero automático (CA)

[Rumbaught 91]

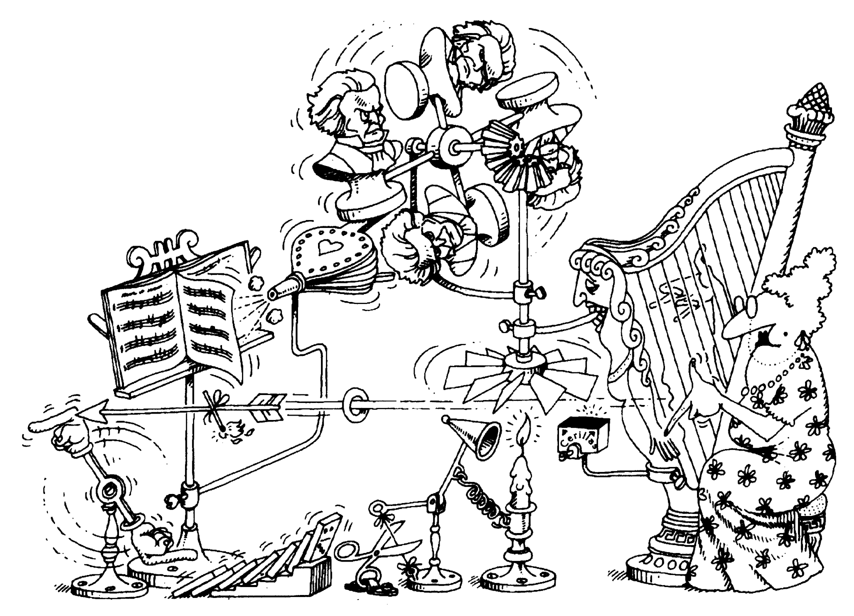


Eliminación de clases innecesarias en el problema del CA.

Mecanismos

Denotan las decisiones estratégicas de diseño respecto a la actividad de colaboración entre muchos tipos diferentes de objetos [Booch 94]

- Identificación de mecanismos
  – Búsqueda de mecanismos
    - Se utilizan escenarios
    - El interfaz de una clase individual es una decisión **táctica**, sin embargo los mecanismos son decisiones **estratégicas** en las que el desarrollador elige entre un conjunto de alternativas influyendo factores como coste, fiabilidad, facilidad de fabricación y seguridad



Los mecanismos son los medios por los cuales los objetos colaboran para proporcionar algún comportamiento de nivel superior.

Diagramas de Interacción

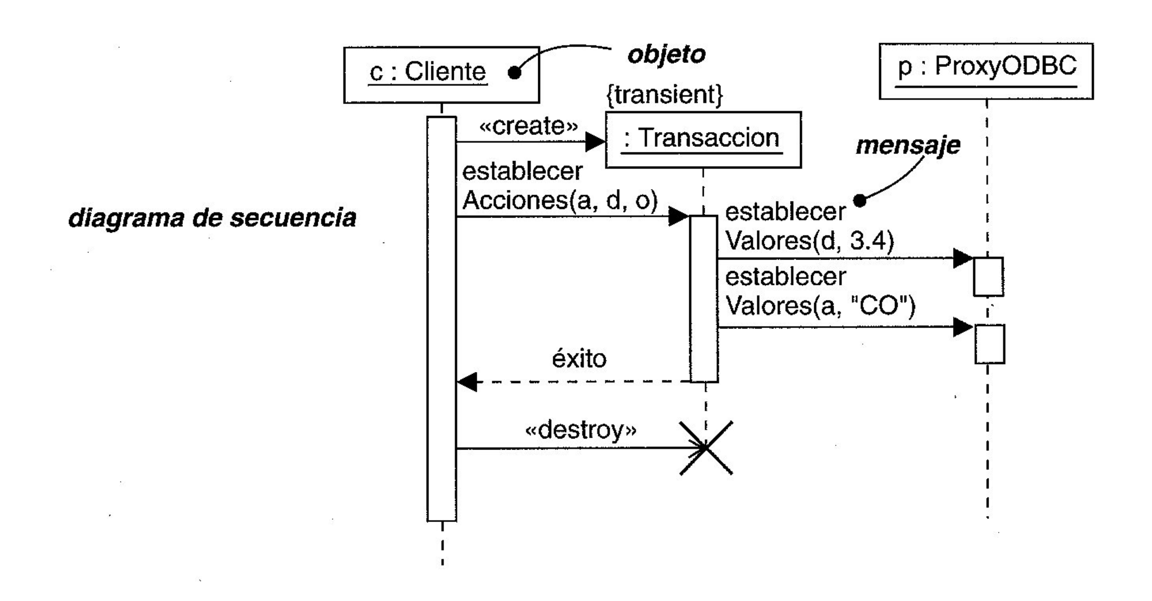
[Booch 99, capítulo 18]

- Son de dos tipos
  - Diagramas de Secuencia
  - Diagramas de Colaboración
- Modelan aspectos dinámicos del sistema
- Un diagrama de interacciones se usa para realizar una traza de la ejecución de un escenario
- A cada escenario le corresponde un diagrama de Interacción, que se suele poner con el mismo código pero con otra letra (E1.1 se corresponde con S 1.1 y C 1.1).
- Un diagrama de interacción muestra una interacción, que consiste en un conjunto de objetos y sus relaciones, incluyendo los mensajes que se pueden enviar entre ellos
- Un diagrama de secuencia es un diagrama de interacción que destaca la **ordenación temporal** de los mensajes
- Un diagrama de colaboración es un diagrama de interacción que destaca la **organización estructural** de los objetos que envían y reciben mensajes
- Un diagrama de interacción contiene
  - Objetos
  - Enlaces
  - Mensajes
- También puede contener
  - Notas
  - Restricciones

Diagramas de Secuencia (I)

[Booch 99, capítulo 18]

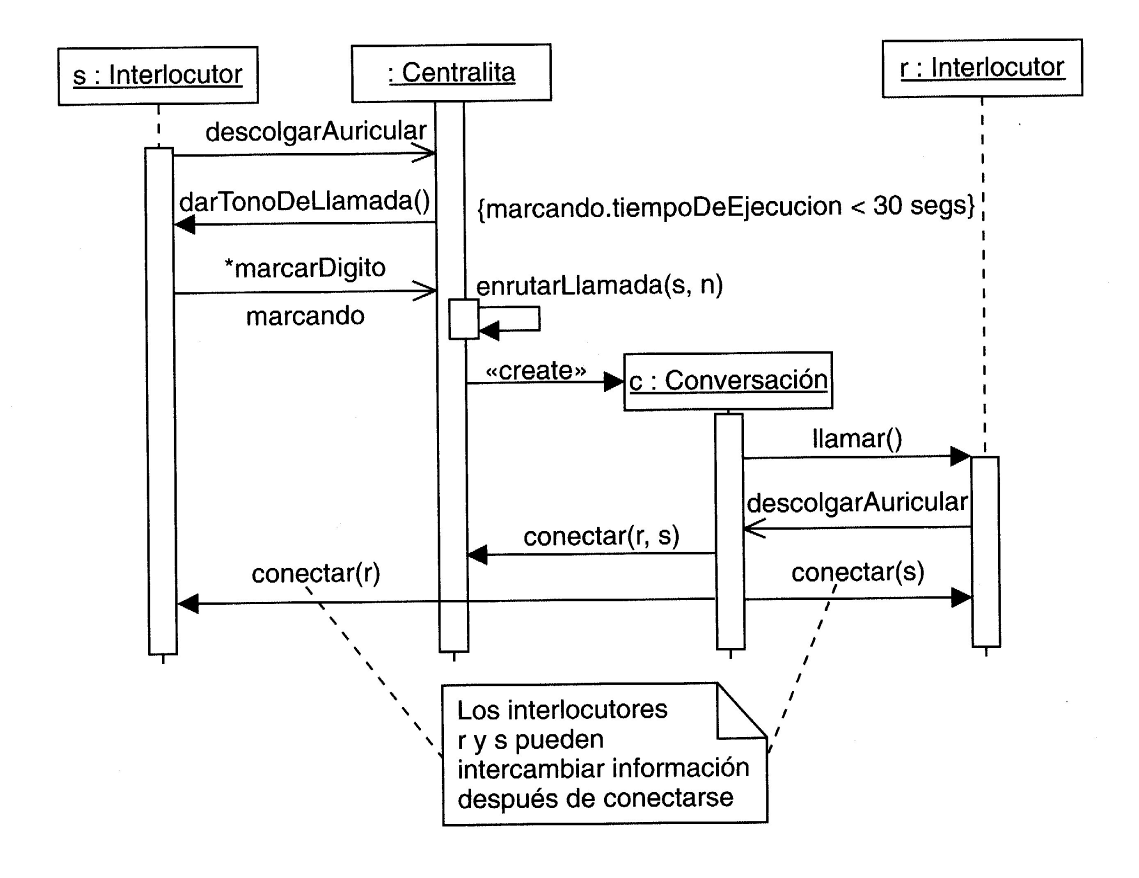
- Se hace un diagrama de secuencia por cada escenario
- Permiten en las fases iniciales de diseño
  - Razonar más en detalle como es el comportamiento de un escenario
  - Obtener nuevas clases y objetos en el escenario (enriquecimiento del diccionario de clases)
  - Detectar cuales son los métodos de las clases, al observar como se relacionan los objetos entre sí para llevar a cabo la tarea encomendada en el escenario
- Se utilizan en las fases de prueba para validar el código
- Si se desea más detalle se utilizan los diagramas de Colaboración



Diagramas de Secuencia (II)

[Booch 99, capítulo 18]

- Los diagramas de secuencia son mejores que los diagramas de colaboración para capturar la semántica de los escenarios en un momento temprano del ciclo de desarrollo.
- Un diagrama de secuencia destaca la ordenación temporal de los mensajes
- Se coloca a la izquierda el objeto que inicia la interacción, y el objeto subordinado a la derecha
- La línea de vida de un objeto es la línea vertical.
  - Es discontinua cuando el objeto no existe
  - Es gruesa y hueca formando un rectángulo cuando existe el objeto y se denomina foco de control
  - La vida del objeto comienza cuando recibe un mensaje estereotipado como <<create>>
  - La vida del objeto finaliza con la recepción de un mensaje estereotipado como <<destroy>> y muestra un aspa indicando el final



Refinamiento de Diagramas de Secuencia (III)

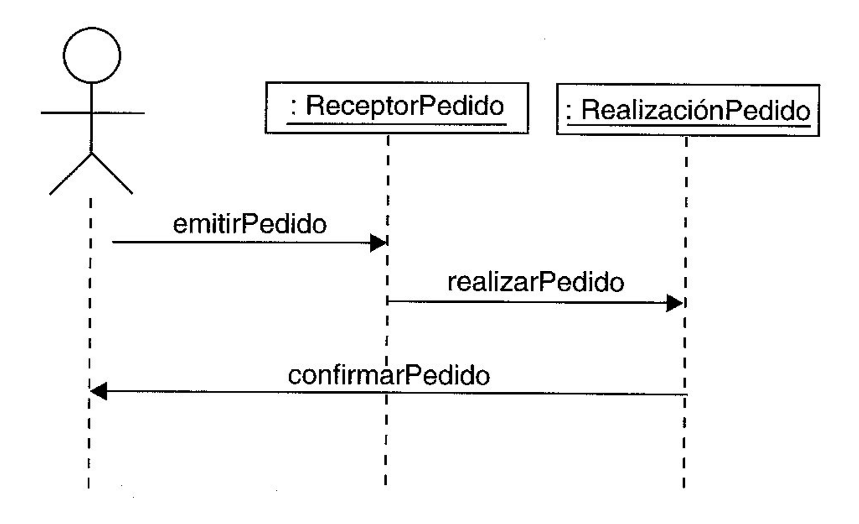


Diagrama de secuencia a un alto nivel de abstracción y poco detallado

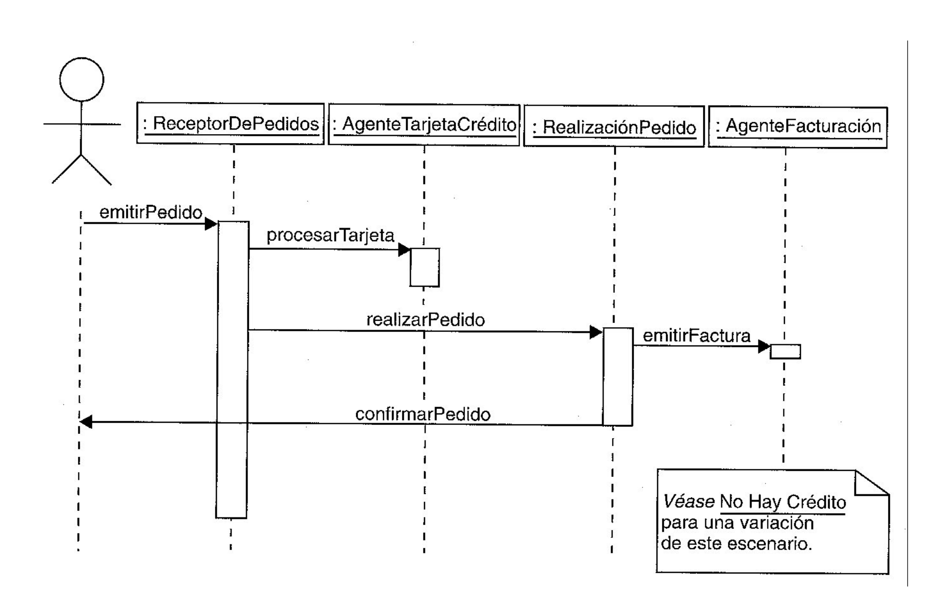
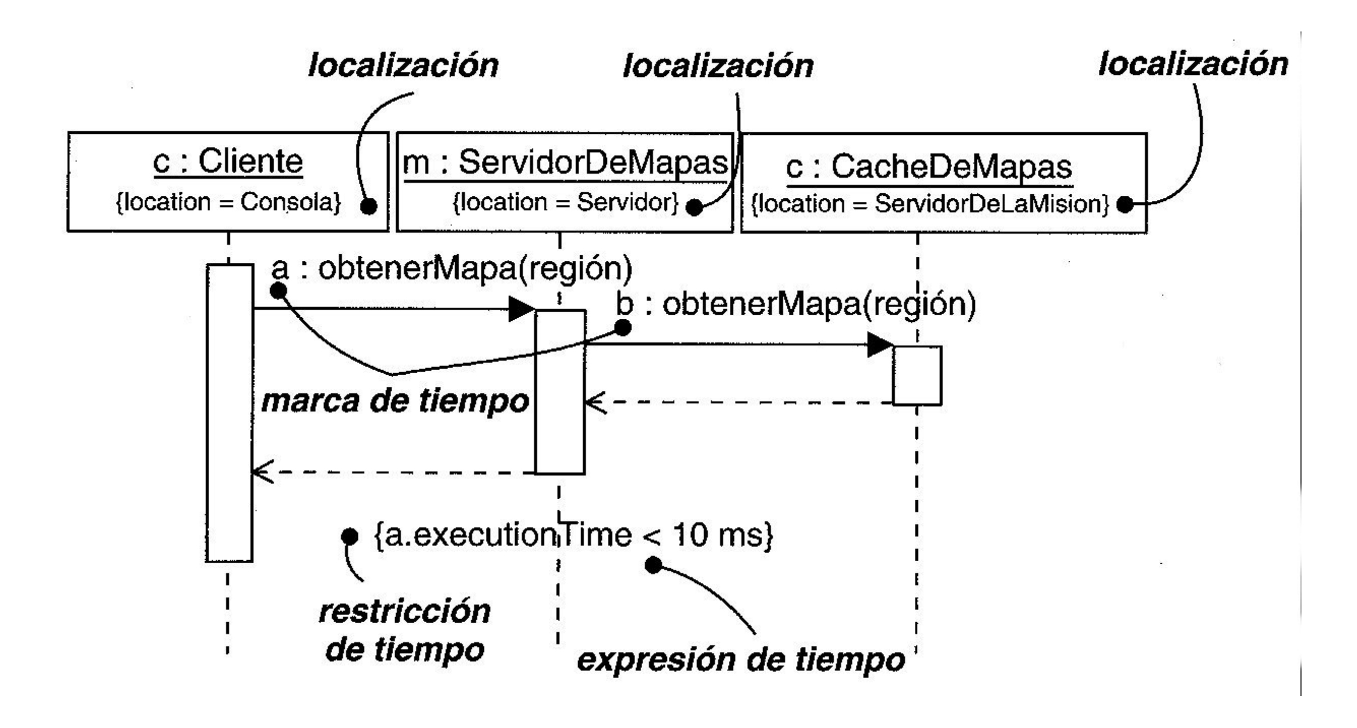


Diagrama de secuencia con menos nivel de abstracción y más detalle

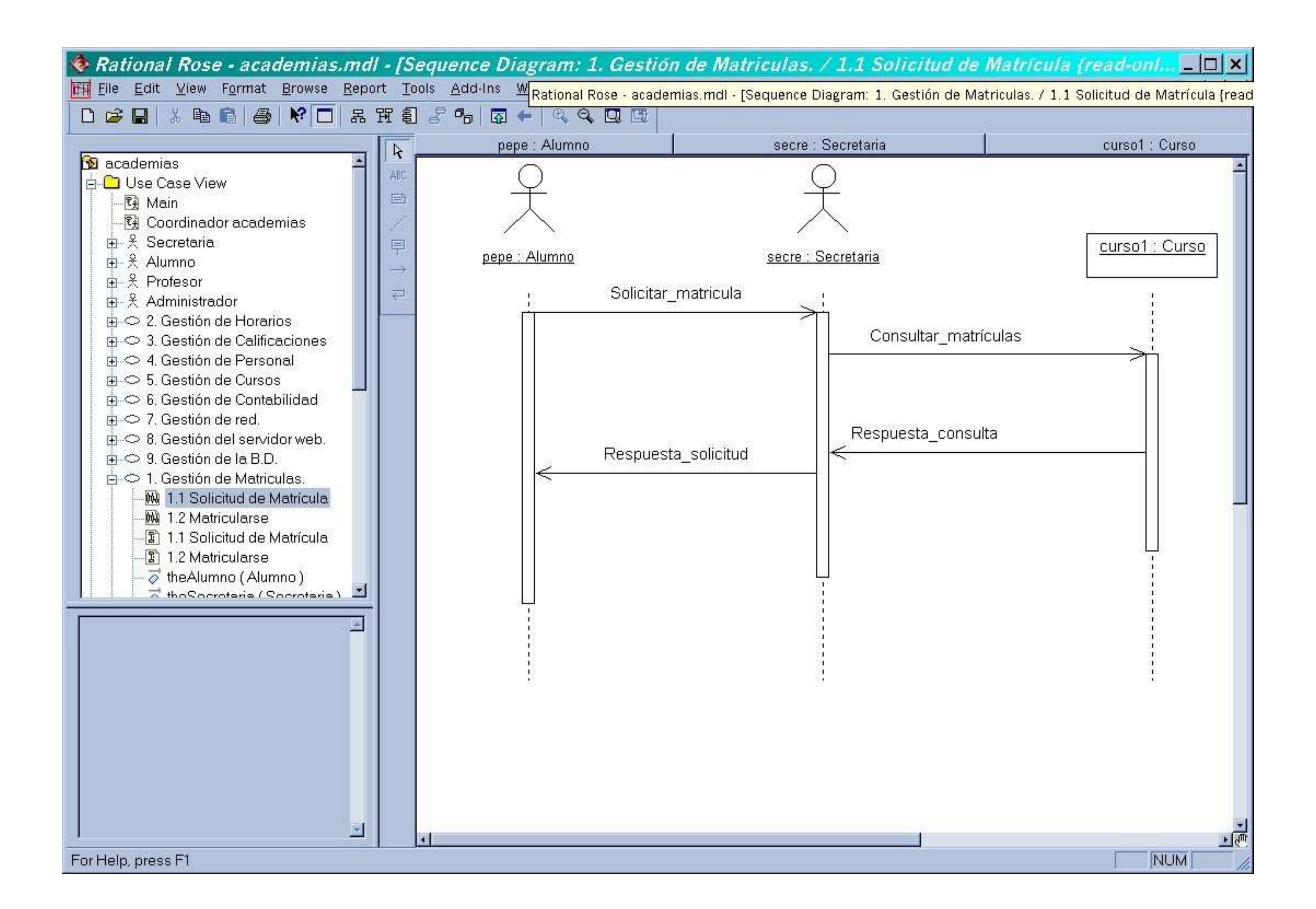
Diagramas de Secuencia (IV)

Restricciones de tiempo y localización

[Booch 99, capítulo 23]



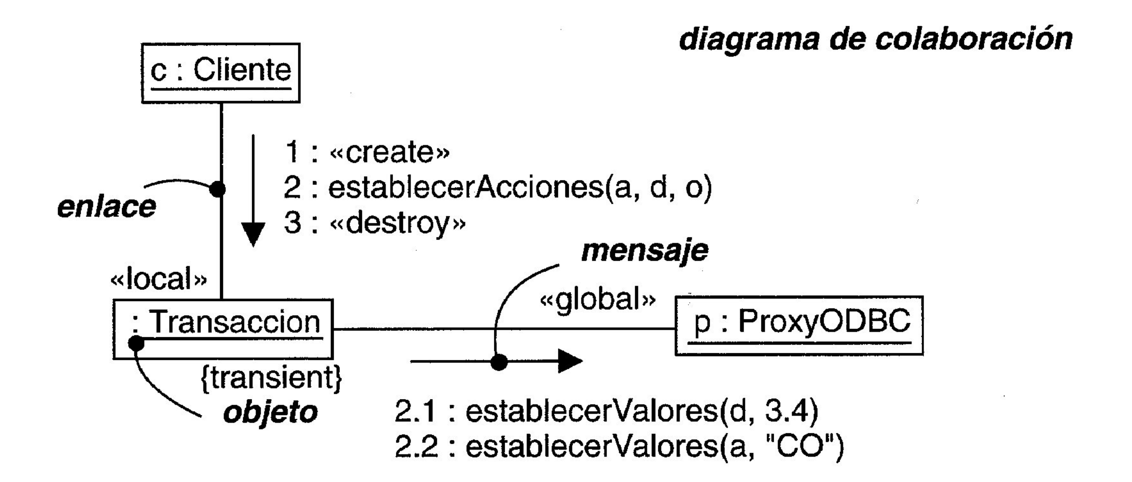
Diagramas de Secuencia (V)
en Rational Rose®



Diagramas de Colaboración (I)

[Booch 99, capítulo 18]

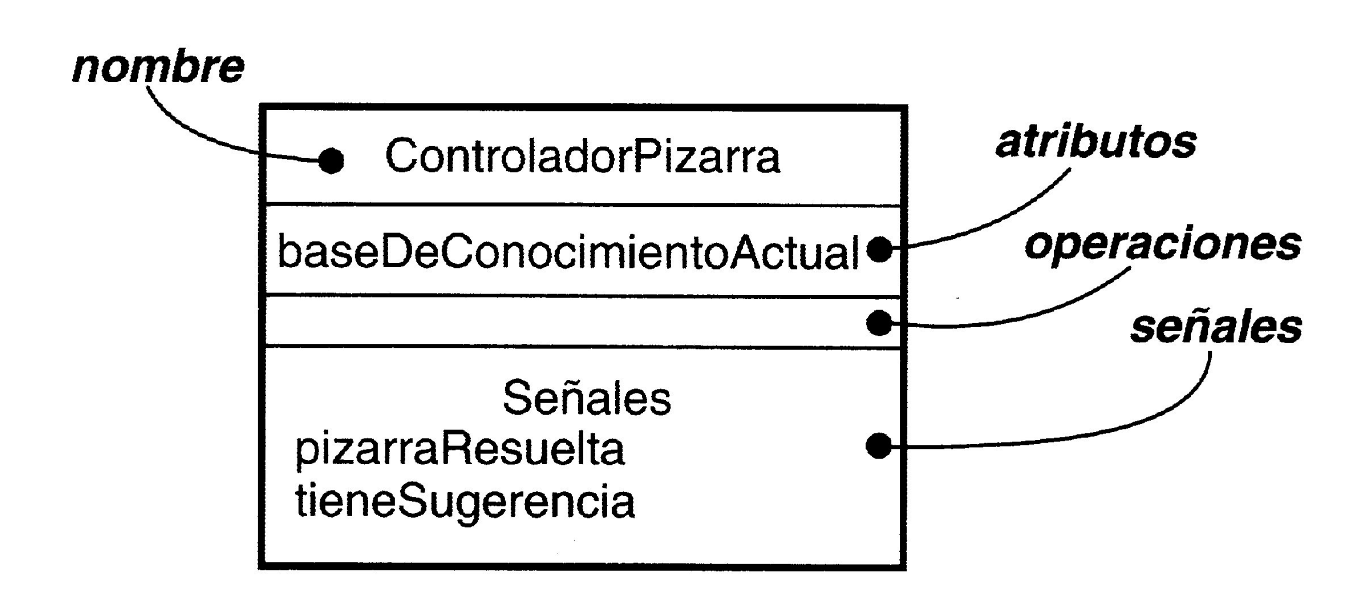
- Permiten profundizar en el nivel de detalle en los diagramas de Secuencia
- Expresan las colaboraciones de los objetos en tiempo de ejecución.
- Dan una visión del flujo de control en el contexto de la organización estructural de los objetos que colaboran.
- Los diagramas de colaboración tienen varias características que los distinguen de los diagramas de secuencia
  - El camino
    - Se puede asociar un estereotipo de camino a cada extremo del enlace
      - `<<local>>` El objeto es visible porque es local al emisor
      - `<<parameter>>` El objeto es visible porque es un parámetro
      - `<<global>>` El objeto es visible porque tiene alcance global
      - `<<self>>` El objeto es visible porque es el emisor del mensaje
  - El número de secuencia indica la ordenación temporal de los mensajes
    - Los mensajes se preceden de un número, que se incrementa secuencialmente (1, 2, 3,...)
    - Para representar el anidamiento se utiliza la numeración decimal de Dewey (dentro del mensaje 2; 2.1 es el primer mensaje dentro de 2; 2.2 el segundo; etc.)
  - Flujos que pueden modelar iteración. Por ejemplo
    - `*[i:=1..n]`
    - o sólo `*` si se quiere indicar iteración sin indicar detalles
  - Flujos que modelan bifurcaciones. Por ejemplo: `[x>0]`
  - Tanto en iteración como en bifurcación UML no impone formato de la expresión entre corchetes; se puede utilizar pseudocódigo o la sintaxis de un lenguaje de programación específico.



Diagramas de Colaboración (II) Modelado de procesos e hilos

[Booch 99, capítulo 22]

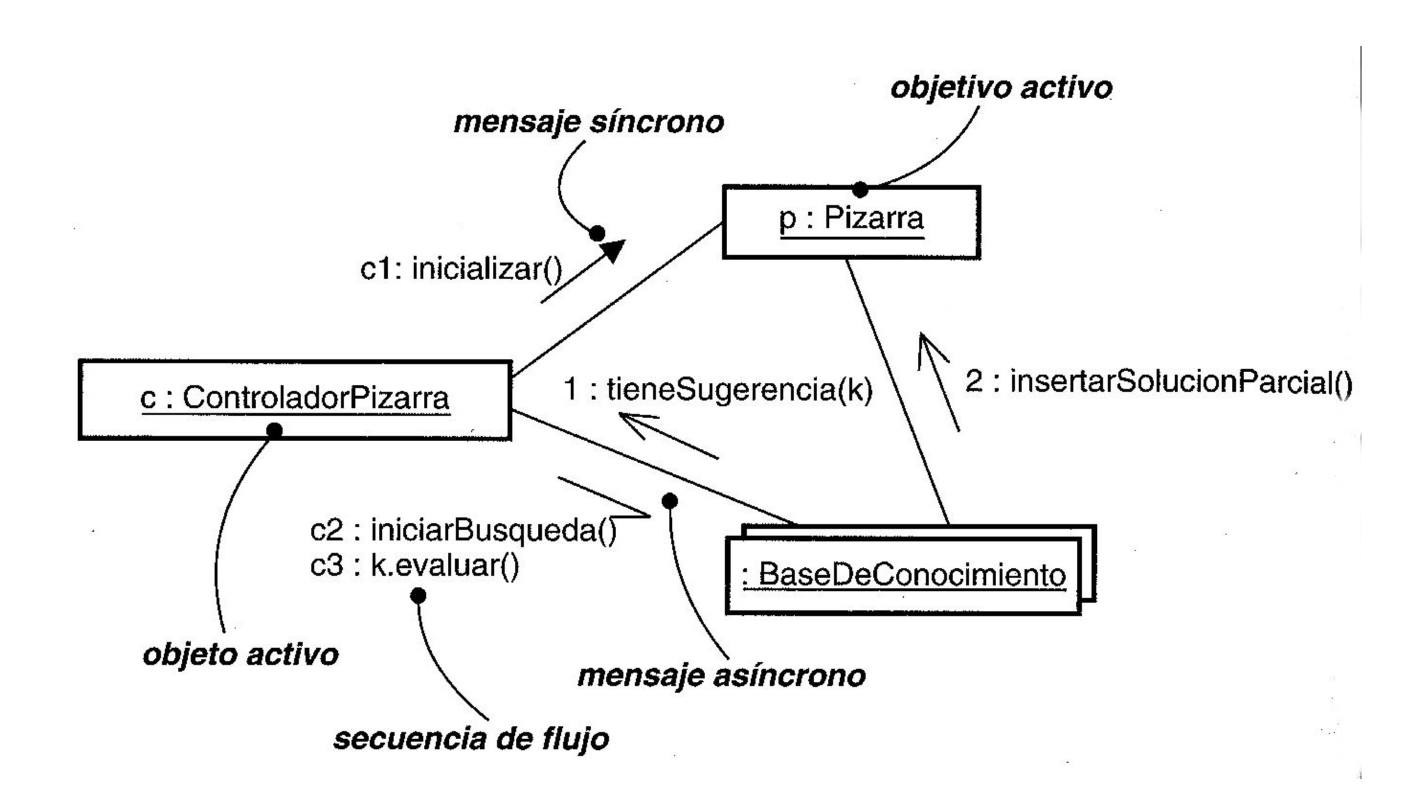
- Un **objeto activo** es un objeto que tiene un proceso o hilo y puede iniciar una actividad de control
- Una **clase activa** es una clase cuyas instancias son objetos activos
  - Una clase activa se representa con un rectángulo con líneas gruesas
- Un **proceso** es un flujo pesado que se puede ejecutar concurrentemente con otro procesos
- Un **hilo** es un proceso ligero que se puede ejecutar concurrentemente con otro hilos dentro del mismo proceso
- Los procesos e hilos se representan como clases activas estereotipadas.
  - UML tiene dos estereotipos estándar para aplicar a las clases activas
    - process
    - thread



Diagramas de Colaboración (III)
Modelado de procesos e hilos
Comunicación

[Booch 99, capítulo 22]

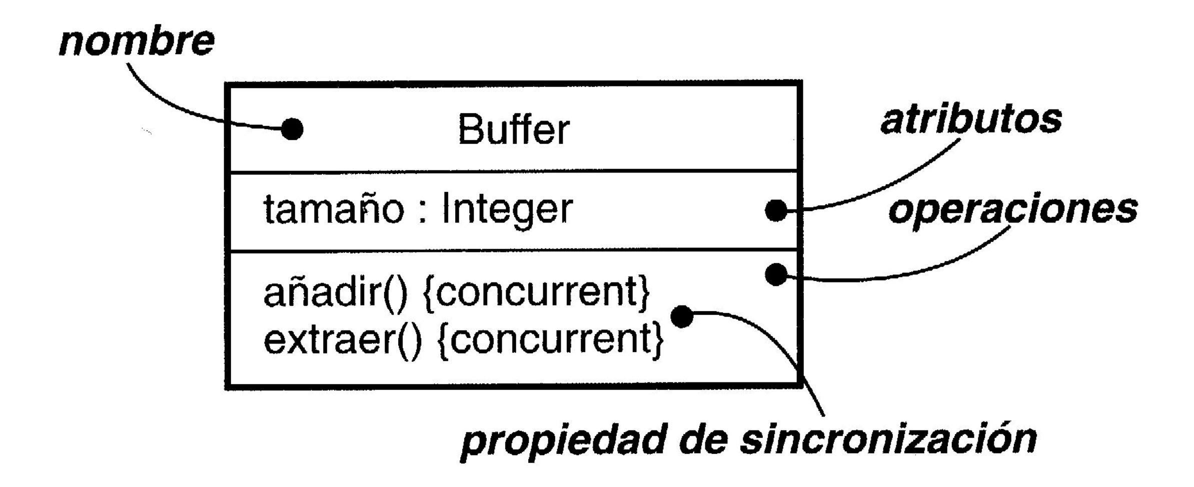
- Mensaje síncrono (flecha completa)
- Mensaje asíncrono (media flecha)



Diagramas de Colaboración (IV)
Modelado de procesos e hilos
Sincronización

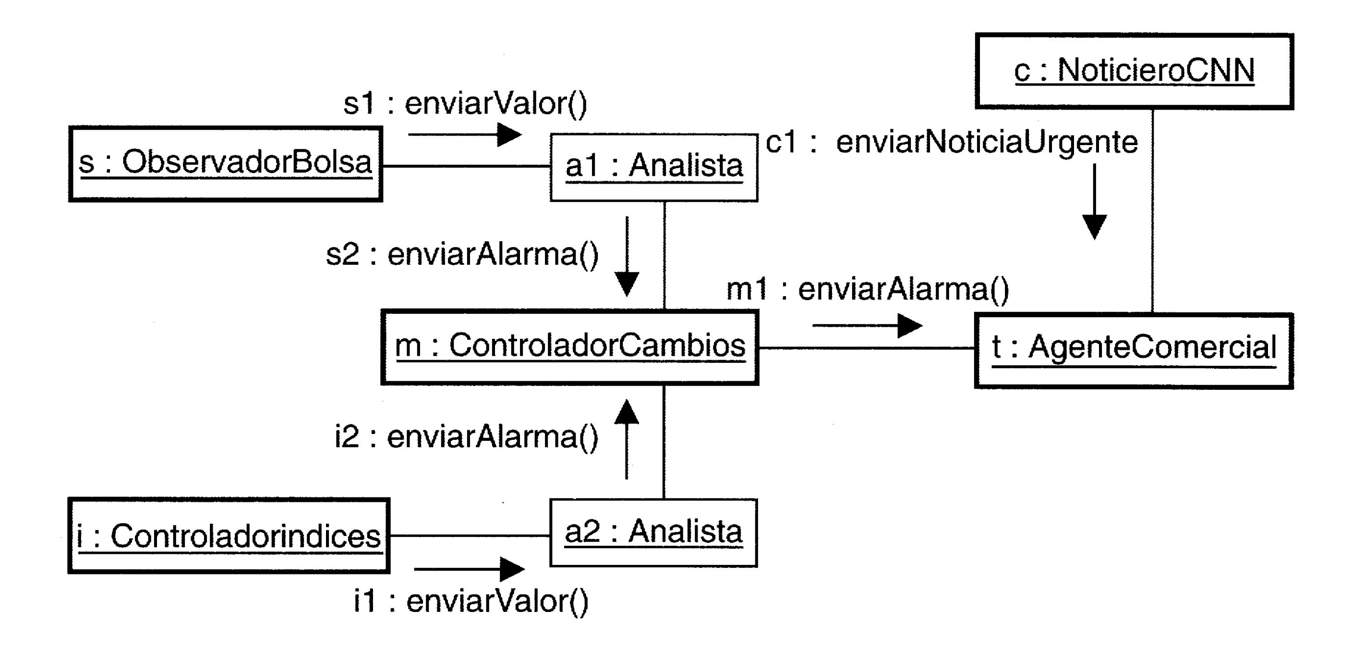
[Booch 99, capítulo 22]

- En UML se pueden modelar los tres enfoques de sincronización
  - Secuencial
  - Con guardas
  - Concurrentemente
- Se utiliza la notación de restricciones de UML asociar propiedades a cada operación



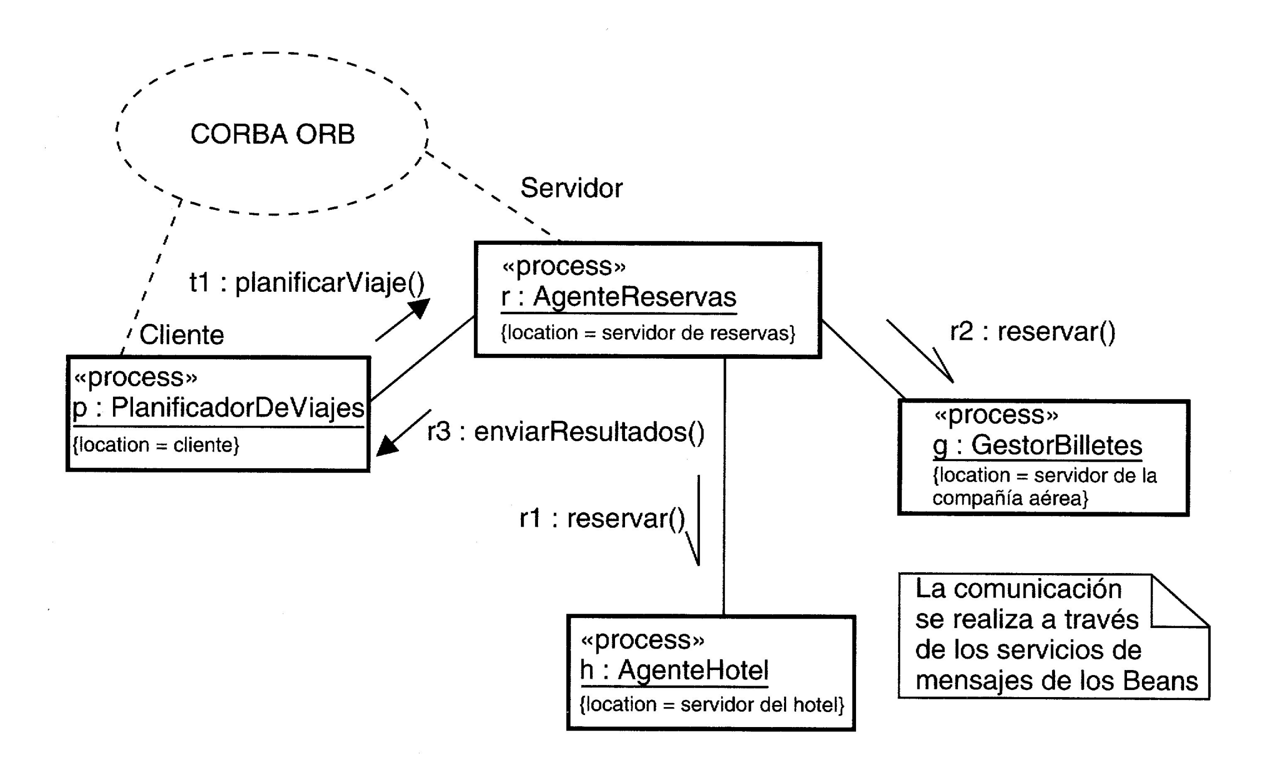
Diagramas de Colaboración (V)
Modelado de procesos e hilos
Modelado de flujos de control múltiples

[Booch 99, capítulo 22]



Diagramas de Colaboración (VI)
Modelado de procesos e hilos
Modelado de comunicación entre procesos

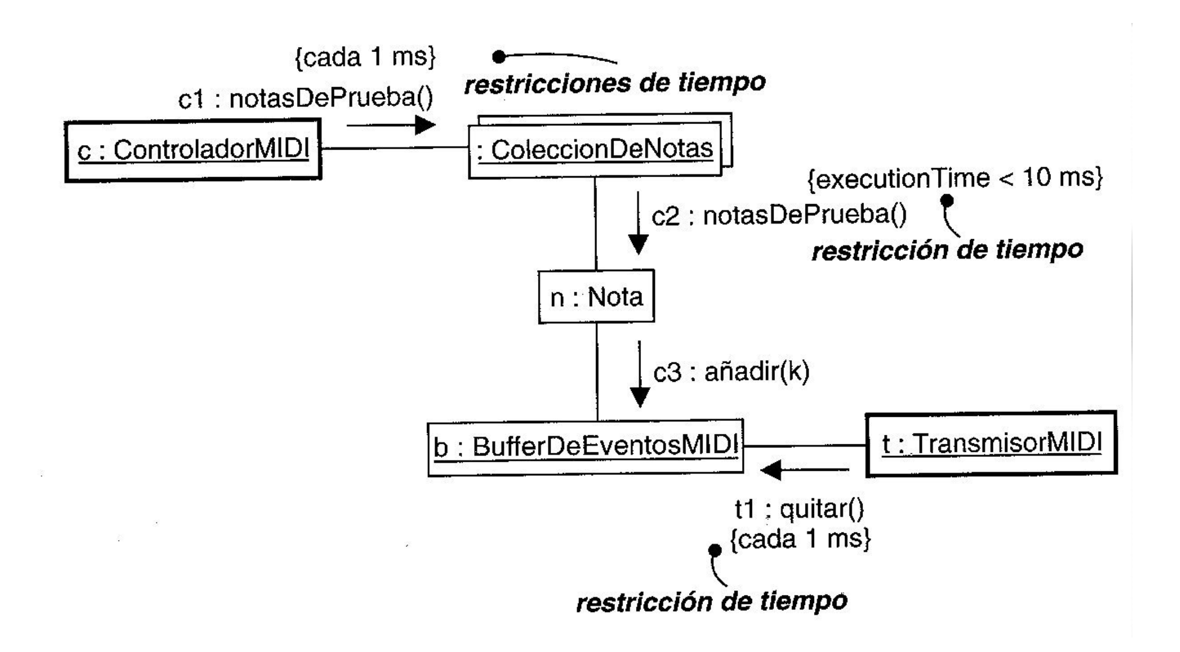
[Booch 99, capítulo 22]



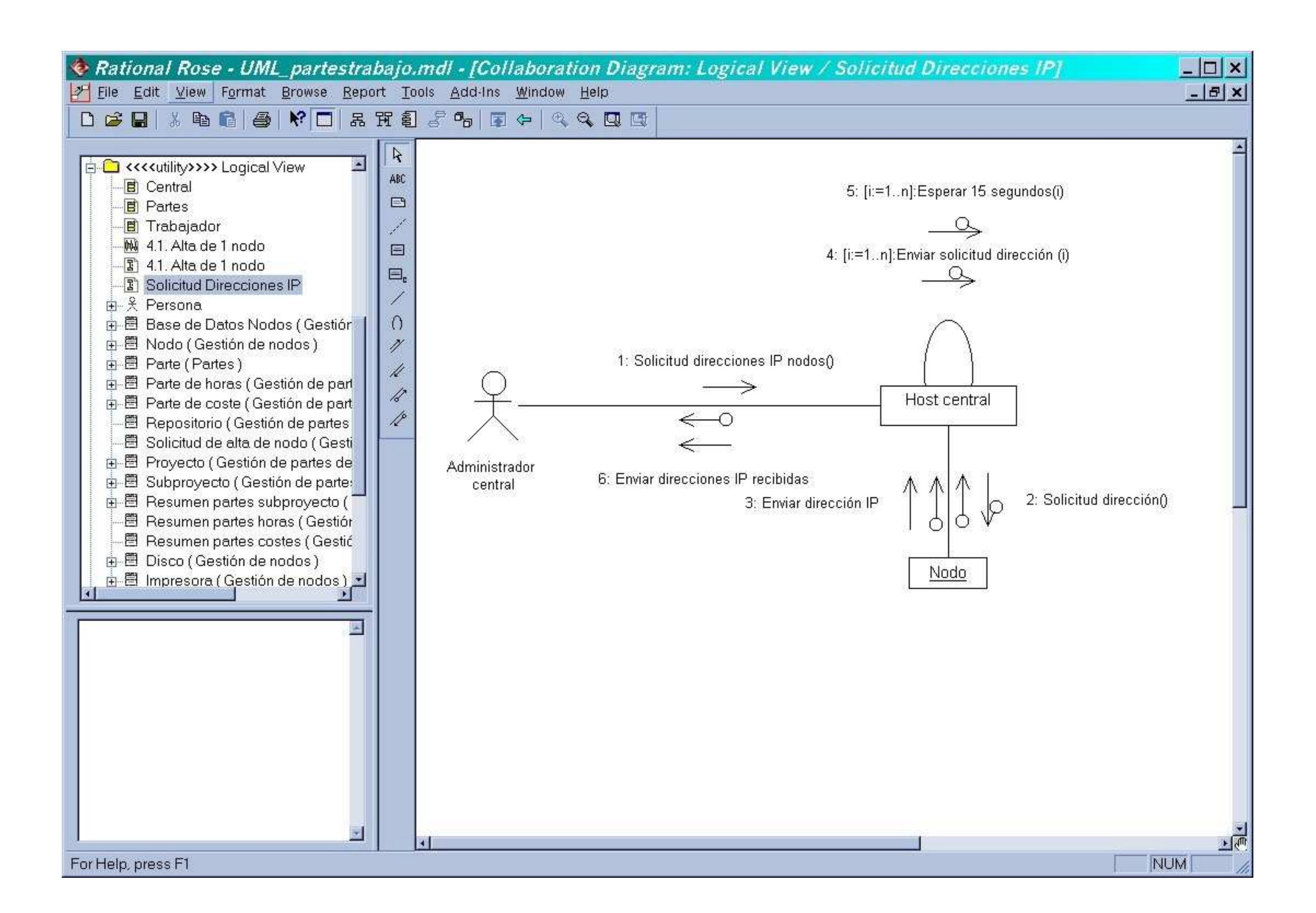
Diagramas de Colaboración (VII)

Restricciones de tiempo

[Booch 99, capítulo 23]



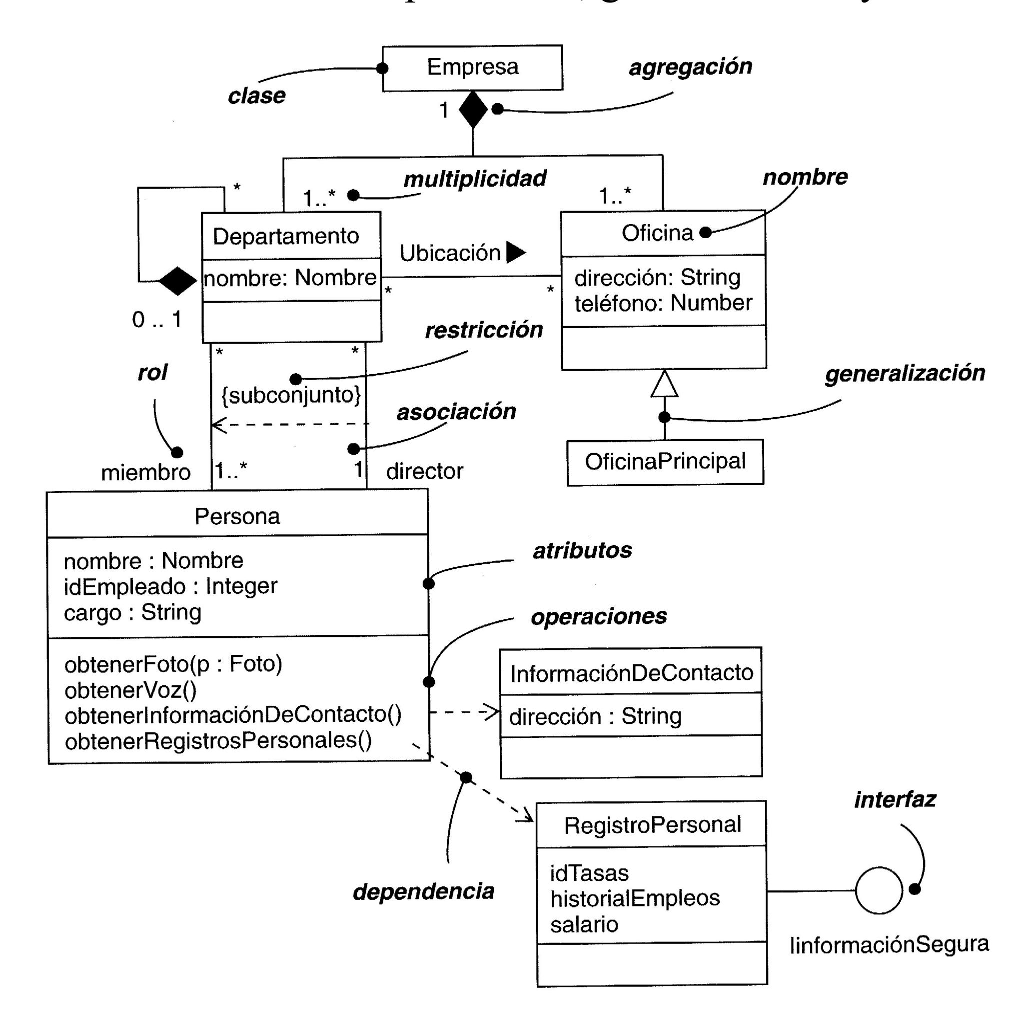
Diagramas de Colaboración (VIII) en Rational Rose®



Diagramas de Clases (I)

[Booch 99, capítulos 8,9]

- Los diagramas de clases se utilizan para modelar la vista de diseño estática de un sistema
- Los diagramas de clases contienen los siguientes elementos
  - Clases
  - Interfaces
  - Colaboraciones
  - Relaciones de dependencia, generalización y asociación



Diagramas de Clases (II)

- Estos diagramas son los más importantes del diseño orientado a objetos, son la *piedra angular* de nuestro diseño.
- Contienen toda la información de todas las clases y sus relaciones con otras clases
- Aunque son los más importantes no se llega a ellos directamente dado que tienen un gran nivel de abstracción dado que contemplan el modelo globalmente sin particularizarse en ningún escenario concreto
- Cuando se construyen los diagramas anteriores (Casos de uso, Secuencia, Colaboración) las herramientas van obteniendo nombres de clase y generando los atributos y operaciones de cada clase siguiendo las indicaciones dadas por las especificaciones de requisitos en los casos de uso y escenarios
- En el momento de hacer el primer diagrama de clases ya se tiene una lista de clases con algunos de sus atributos y operaciones. Sin embargo es necesario reflexionar y abstraer sobre la organización de esas clases estudiando las relaciones de herencia, agregación, etc.
- El diagrama de clases se refinará en las sucesivas iteraciones del modelo

Clases y objetos en UML

Clase

Objetos

Point

x: Real
y: Real

rotate (angle: Real)
scale (factor: Real)

p1: Point

x = 3.14
y = 2.718

:Point

x = 1
y = 1.414

Los compartimentos de una clase pueden tener nombres

Reservation

operations

guarantee()
cancel ()
change (newDate: Date)

responsibilities

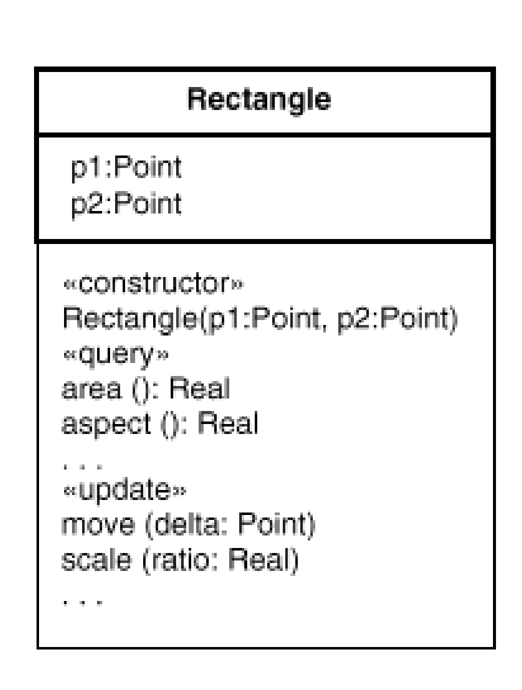
bill no-shows
match to available rooms

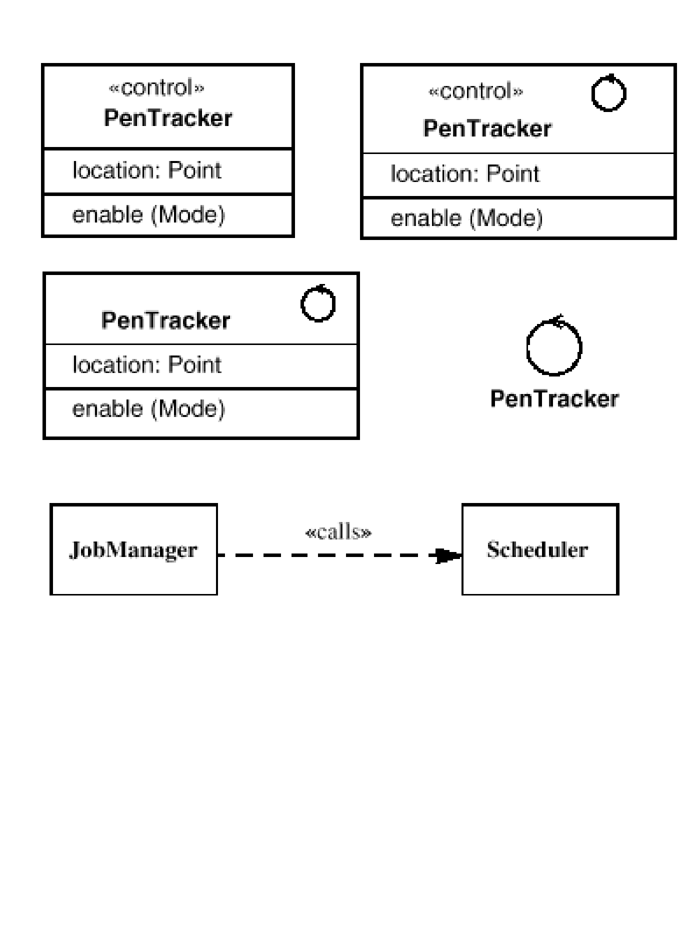
exceptions

invalid credit card

Estereotipos en UML

• Un estereotipo es una forma de clasificar las clases a alto nivel
• Los estereotipos se muestran con un texto entre doble ángulo << y >>, por ejemplo: *<<control>>*
• Los estereotipos también se pueden definir con un icono
• Muchas extensiones del núcleo de UML se pueden describir mediante una colección de estereotipos

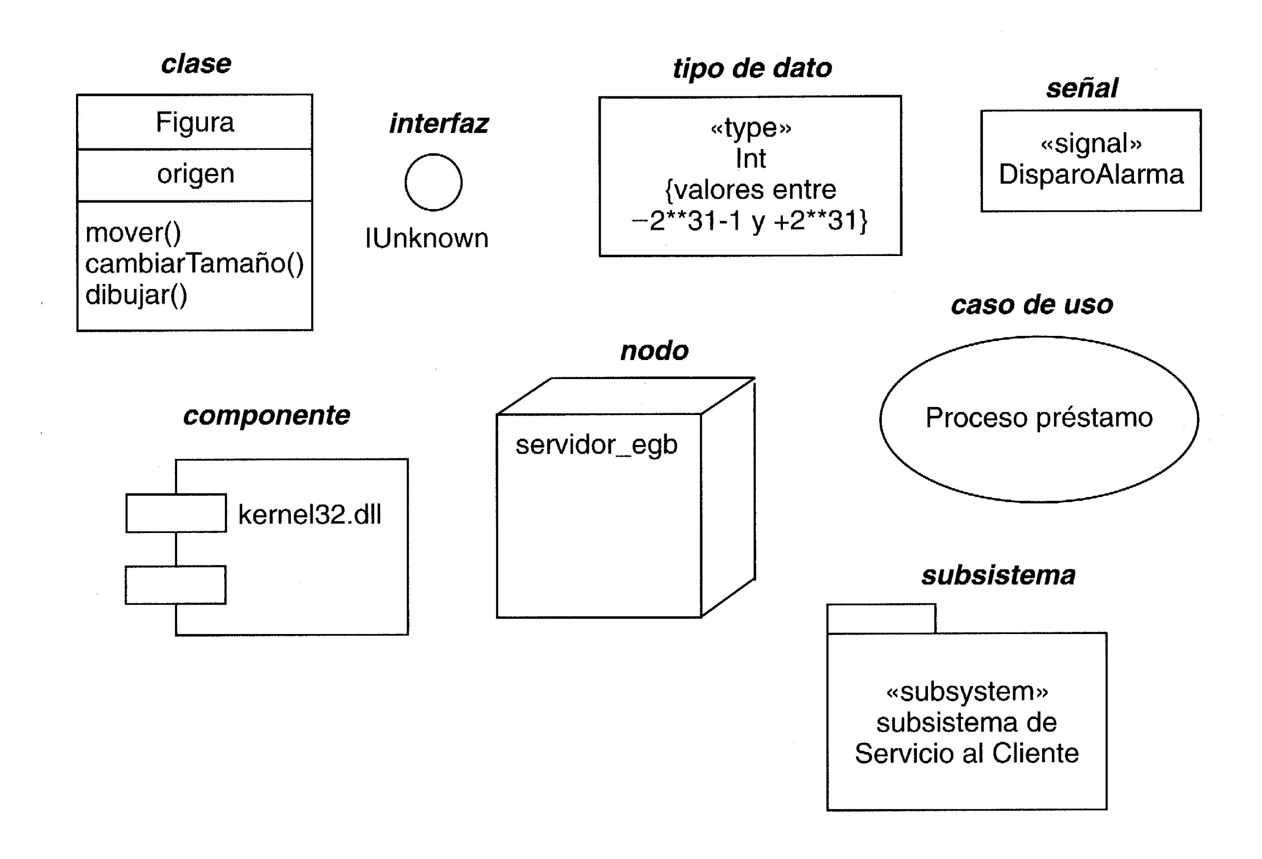




Clasificadores (I)

[Booch 99, capítulo 9]

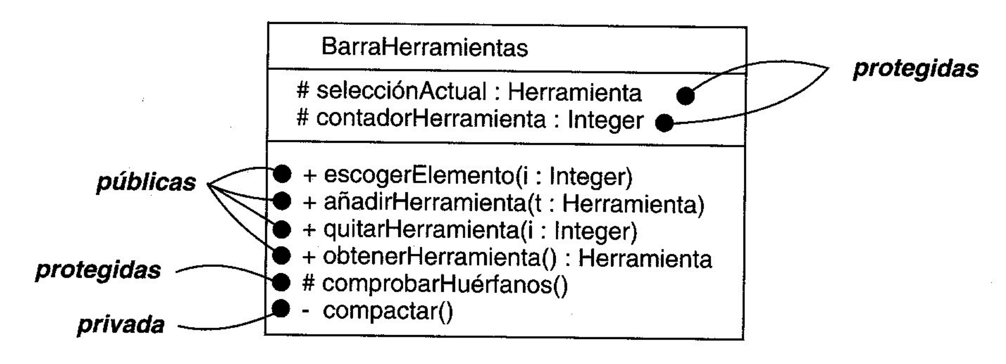
- Un clasificador es un mecanismo que describe características estructurales y de comportamiento
- Tipos de clasificadores:
  - Clase
  - Interfaz
  - Tipo de dato
  - Señal
  - Componente
  - Nodo
  - Caso de Uso
  - Subsistema



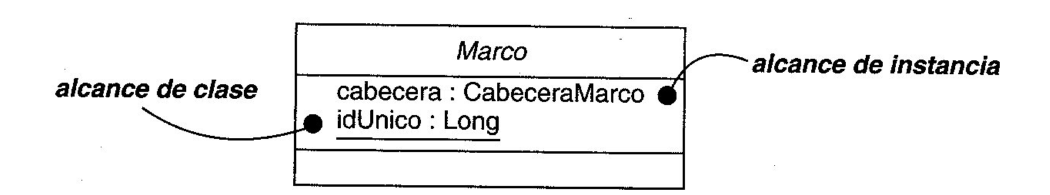
Clasificadores (II)

[Booch 99, capítulo 9]

- Visibilidad de atributos y operaciones de un clasificador
  - `public`
  - `protected`
  - `private`



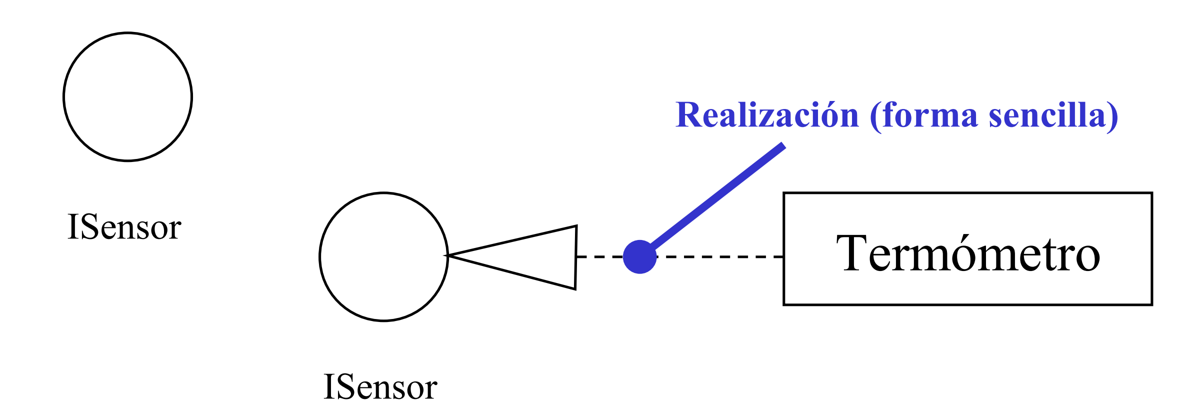
- Alcance de atributos y operaciones de un clasificador
  - `instancia`. Cada instancia del clasificador tiene su propio valor para la característica
  - `clasificador`. Sólo hay un valor de la característica para todas las instancias (static en C++ y Java)

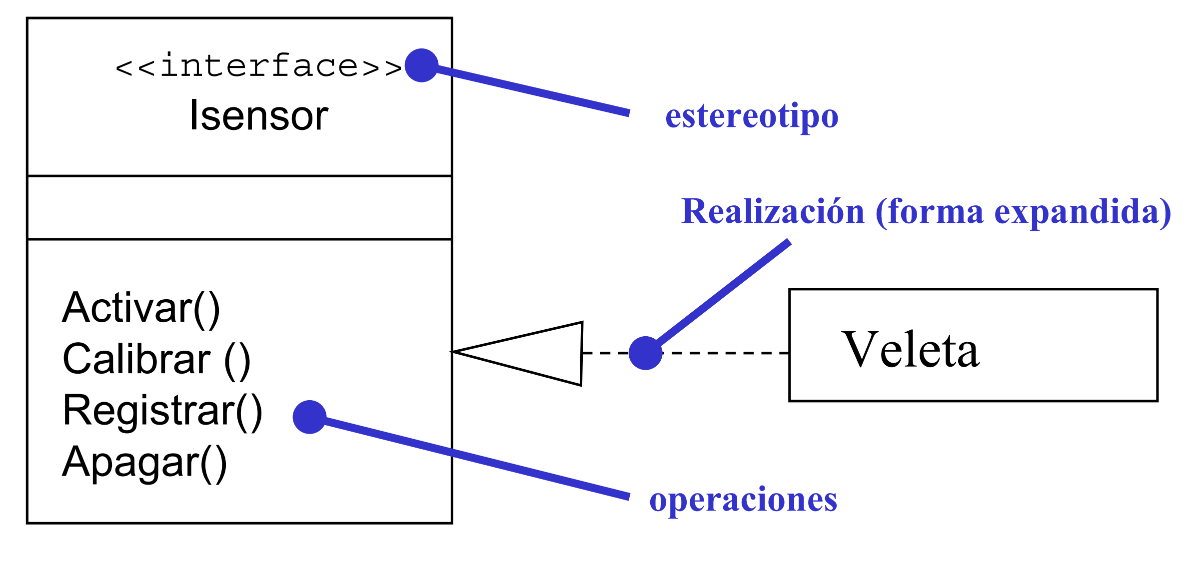


Interfaces

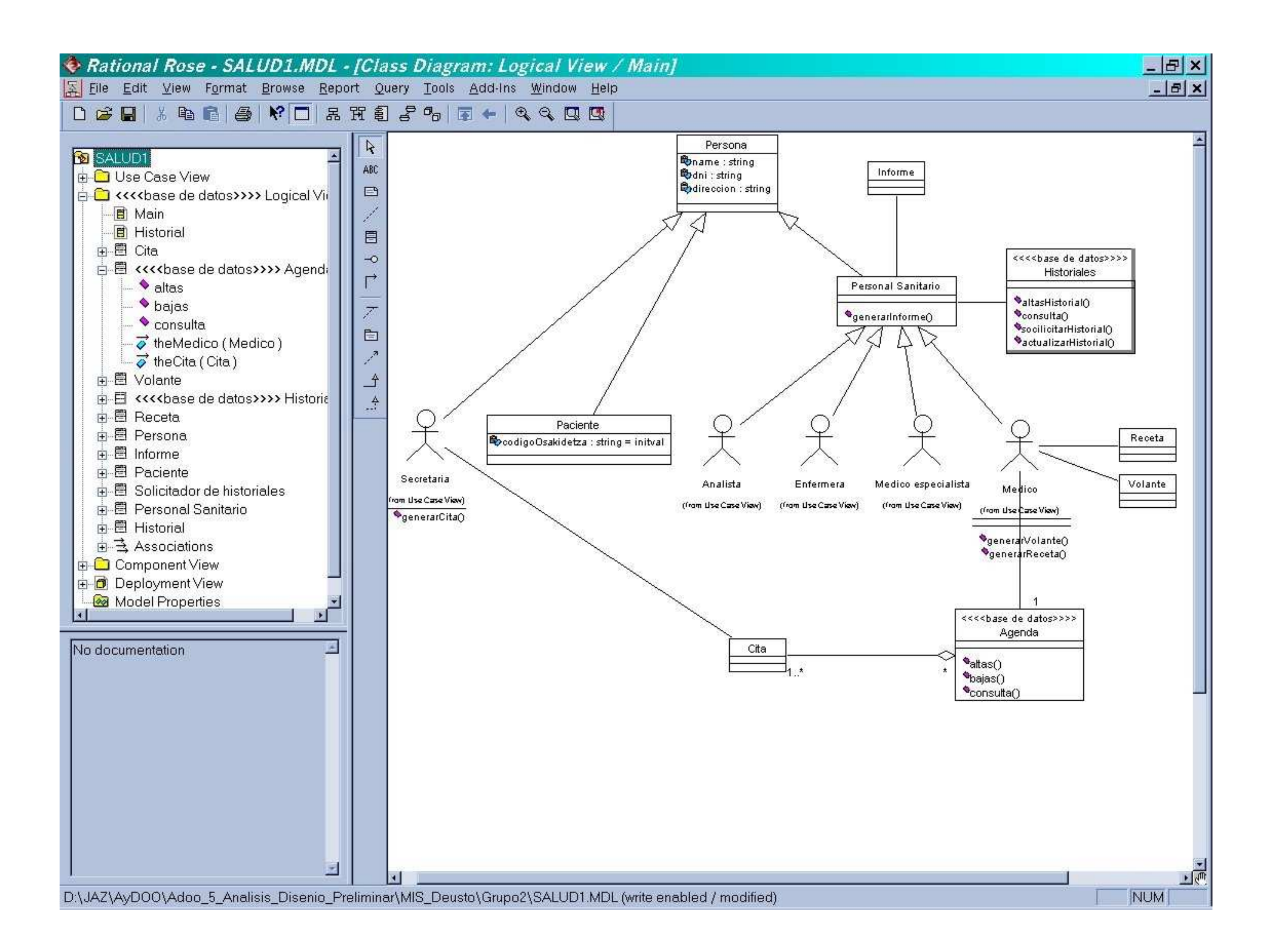
[Booch 99, capítulo 11]

- Una interfaz es una colección de operaciones que se usa para especificar un servicio de una clase o de un componente
- Una interfaz no tiene atributos
- Gráficamente una interfaz se representa por un círculo
- De forma expandida se puede ver como una clase con el estereotipo `<<interface>>`





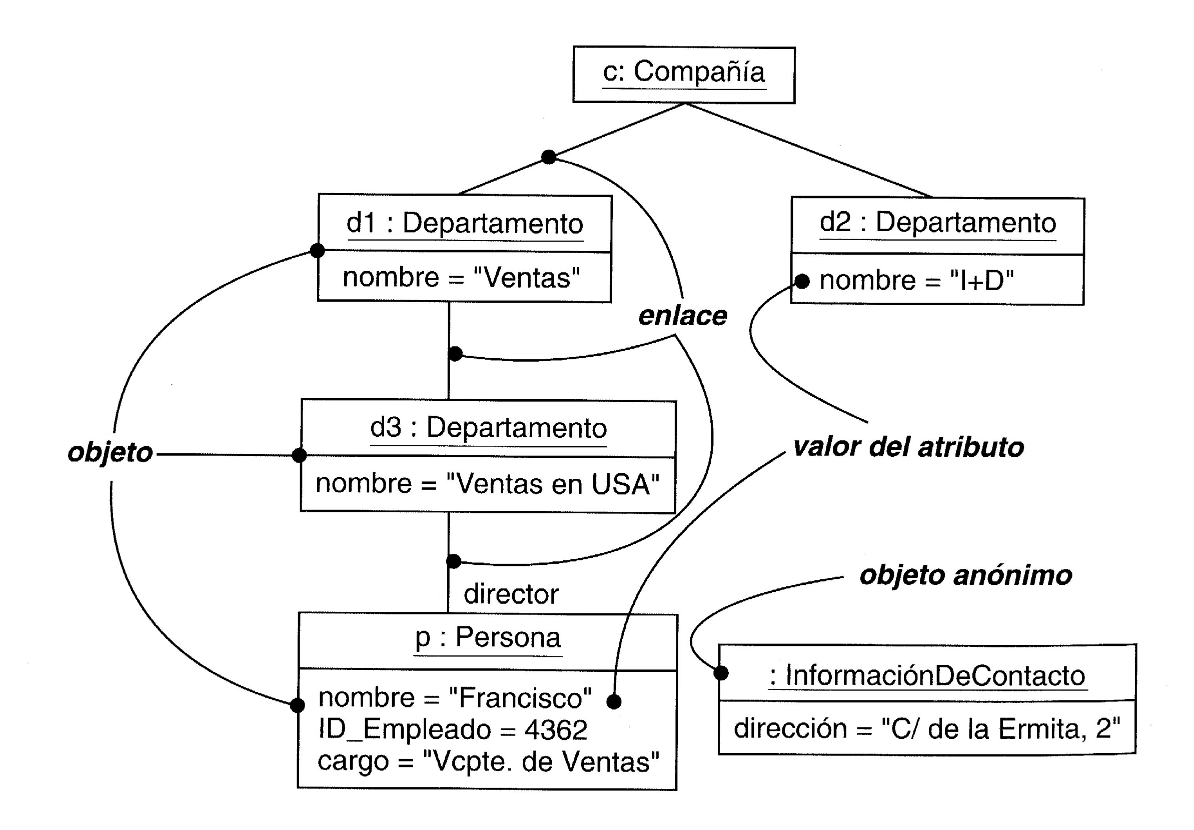
Diagramas de Clases en Rational Rose®



Diagramas de objetos

[Booch 99, capítulo 14]

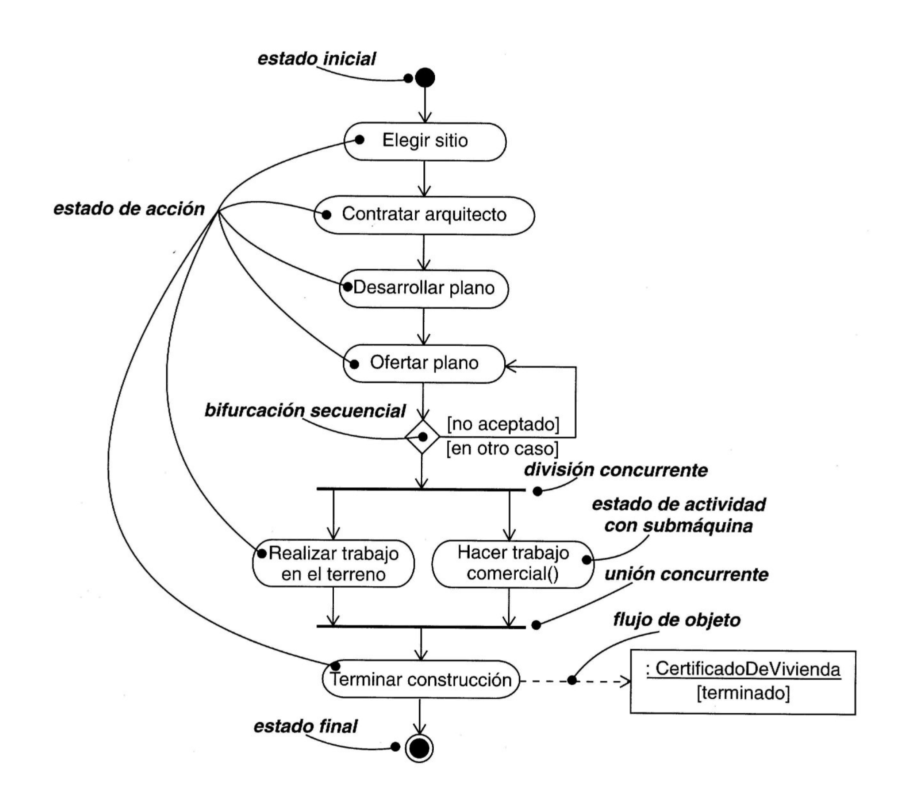
- Muestra un conjunto de objetos y sus relaciones
- Los diagramas de objetos representan instantáneas de instancias de los elementos encontrados en los diagramas de clases
- Estos diagramas cubren la vista de diseño estática o la vista de procesos estática de un sistema como lo hacen los diagramas de clases, pero desde la perspectiva de casos reales o prototípicos
- Muestran una especie de fotograma de un instante en tiempo de ejecución.
- Se realizan para mostrar momentos críticos del sistema o para aclarar detalles que pueden quedar confusos.
- Evidentemente no se hacen para todos los instantes de una ejecución



Diagramas de actividades

[Booch 99, capítulo 19]

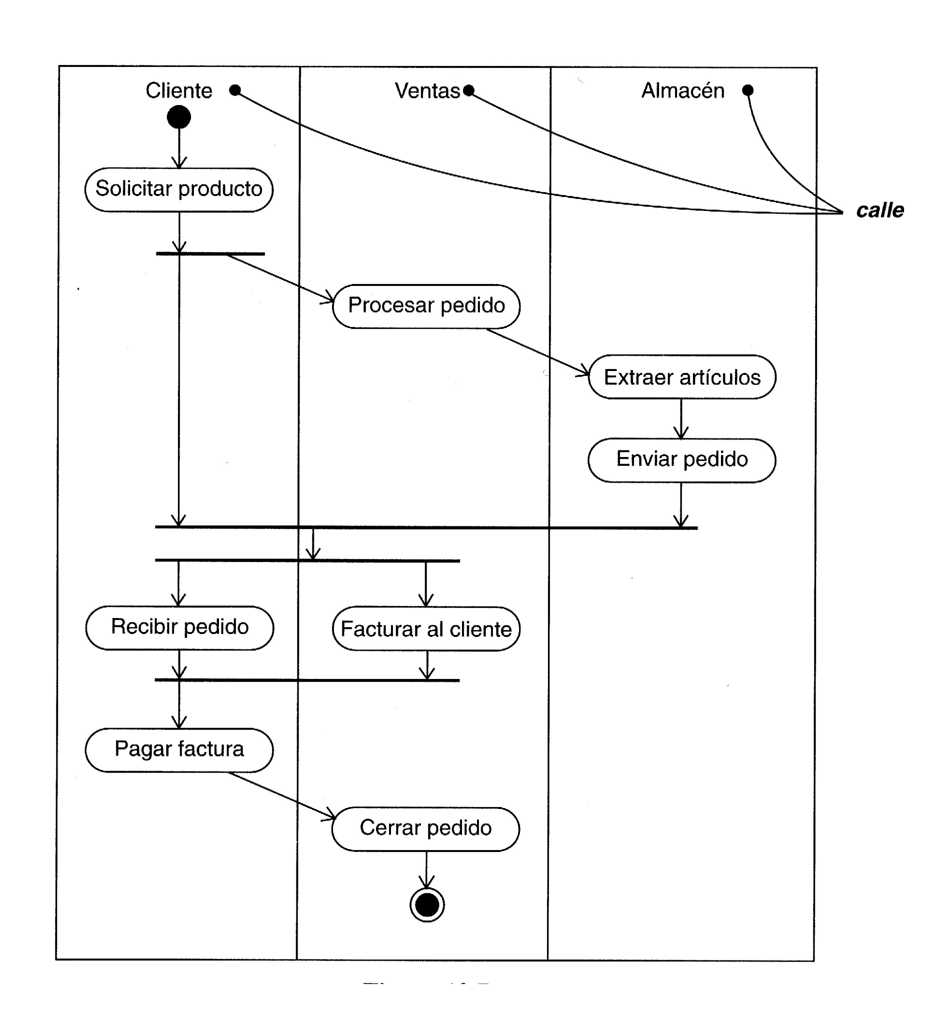
- Los diagramas de actividades son uno de los cinco diagramas que modelan aspectos dinámicos del sistema
- Un diagrama de actividades muestra el flujo de actividades
- Una actividad es una ejecución no atómica en curso dentro de una máquina de estados
- Las actividades producen finalmente una acción, que está compuesta de computaciones atómicas ejecutables que producen un cambio en el estado del sistema o la devolución de un valor.
- Un diagrama de actividad contiene:
  - Estados de actividad y estados de acción
  - Transiciones
  - Objetos
  - Restricciones



Diagramas de actividades
Calles (Swimlanes)

[Booch 99, capítulo 19]

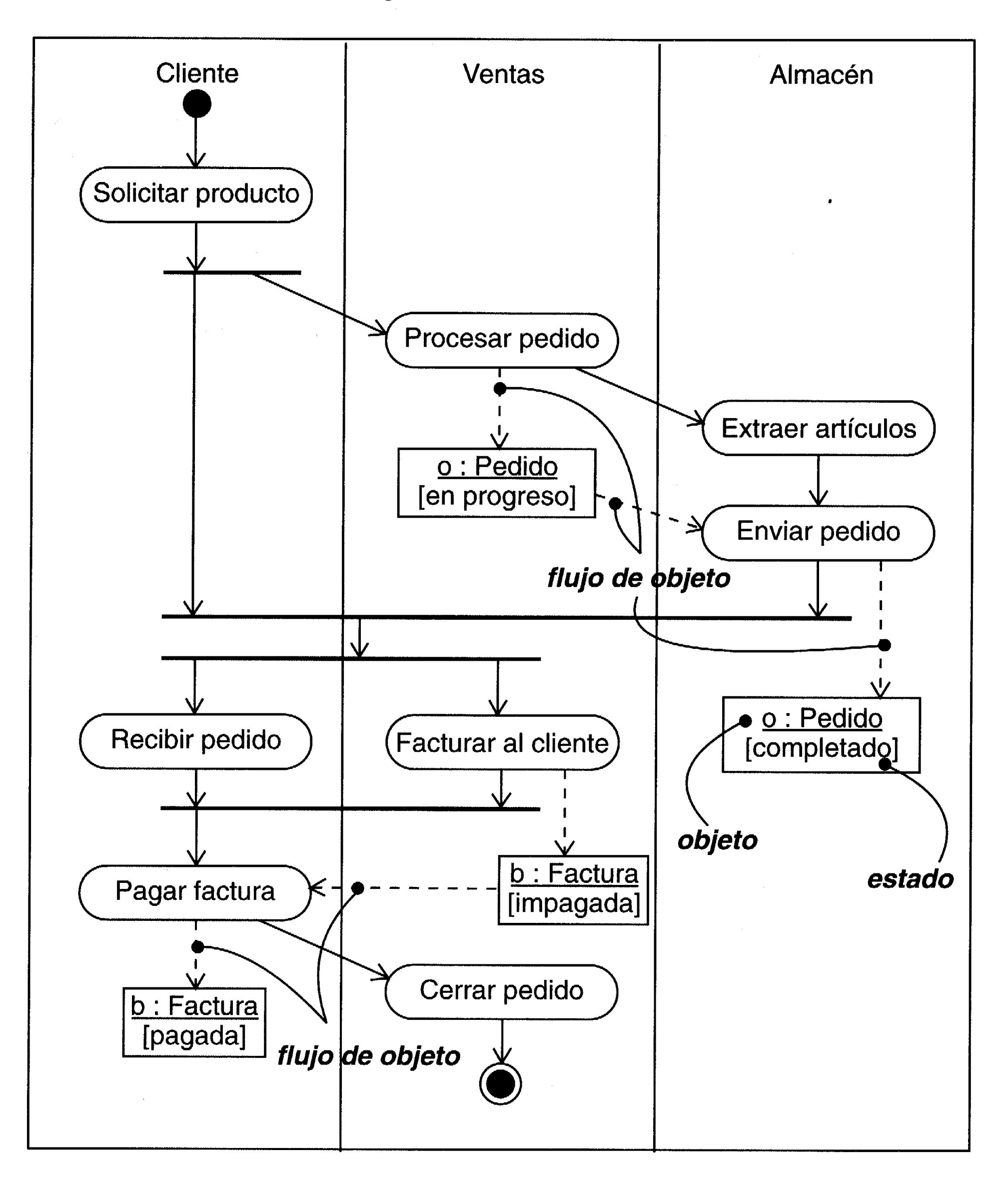
- Permiten modelar flujos de trabajo de procesos de organizaciones separándolos en grupos denominados **calles** (*swinlanes*)
- Cada grupo representa la parte de la organización responsable de esas actividades



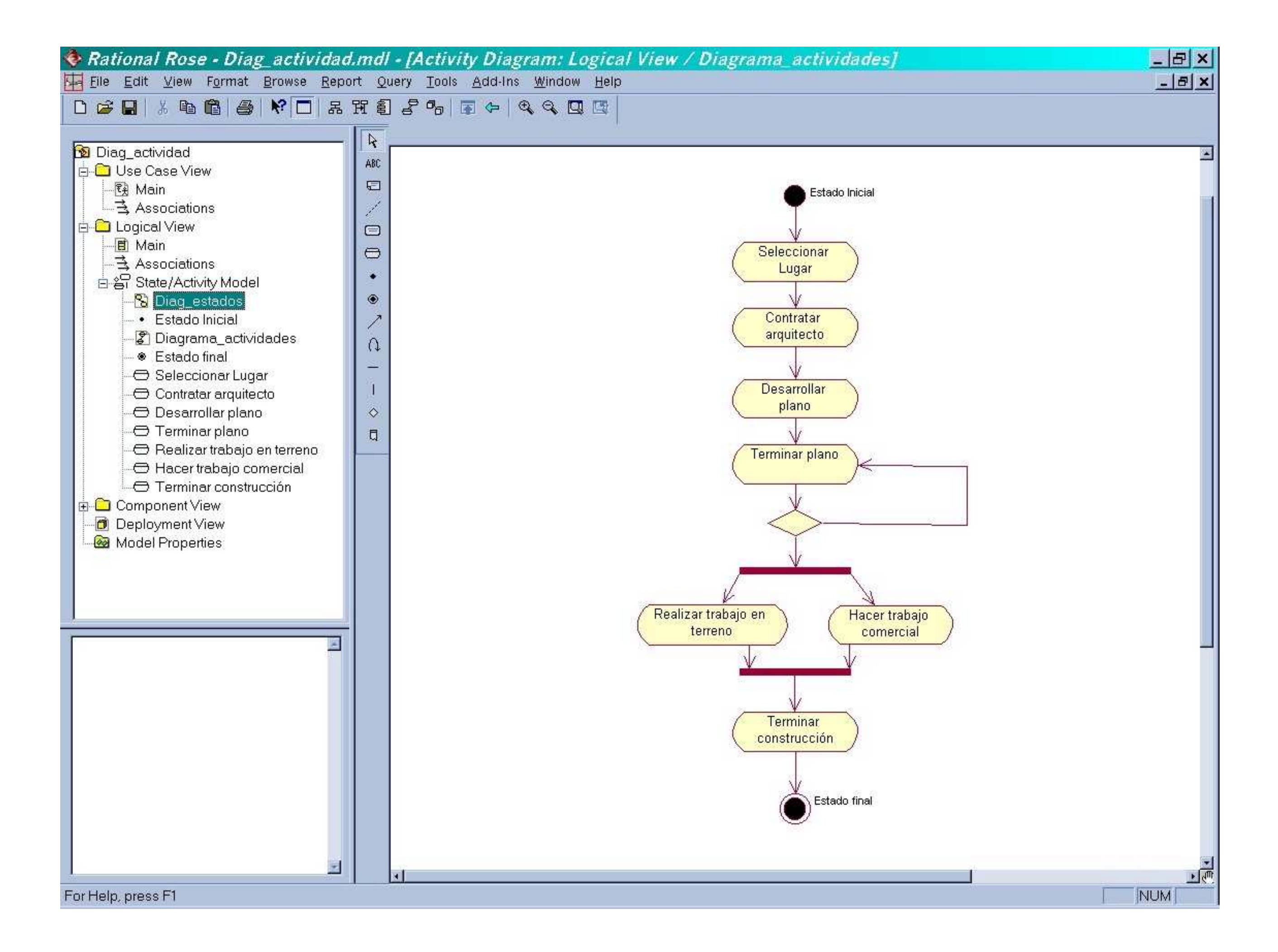
Diagramas de actividades
Flujo de objetos

[Booch 99, capítulo 19]

- Los diagramas de actividad pueden tener un flujo de control con objetos
- Pueden mostrarse también como cambian los valores de los atributos de los objetos



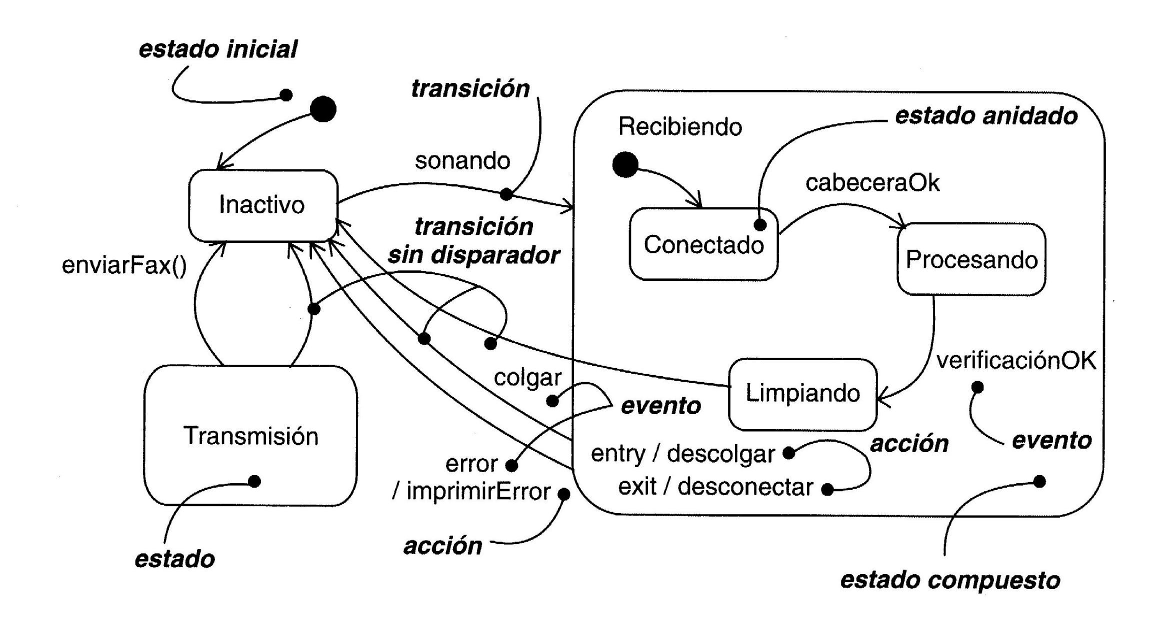
Diagramas de Actividades en Rational Rose®



Diagramas de estados

[Booch 99, capítulo 24]

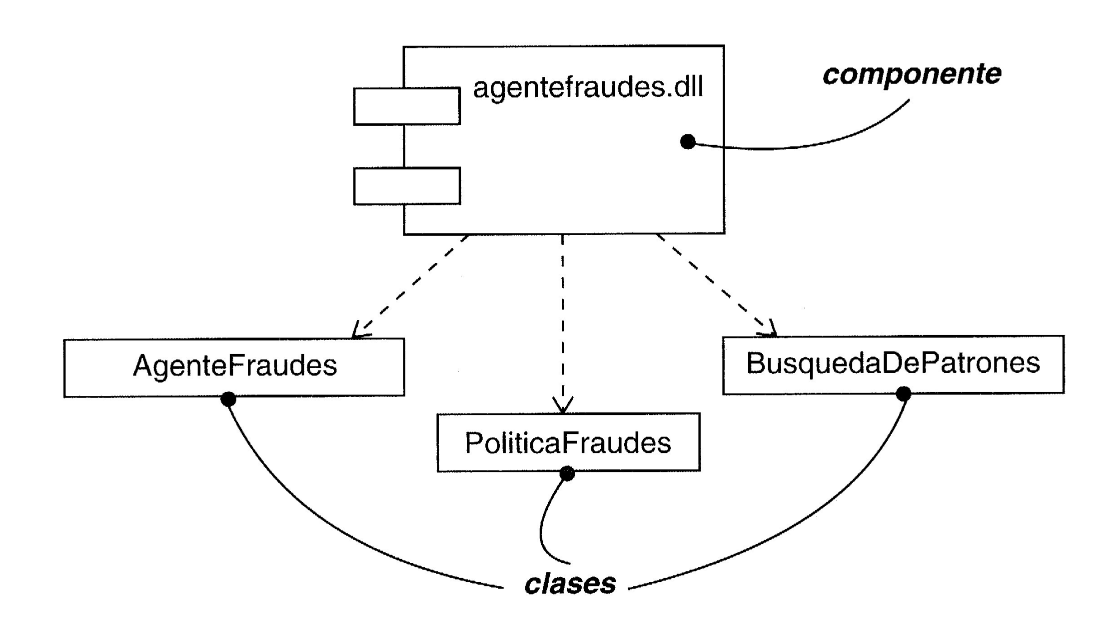
- Se utilizan para modelar aspectos dinámicos de un sistema
- Un diagrama de estados muestra una máquina de estados.



Diagramas de componentes

[Booch 99, capítulo 25]

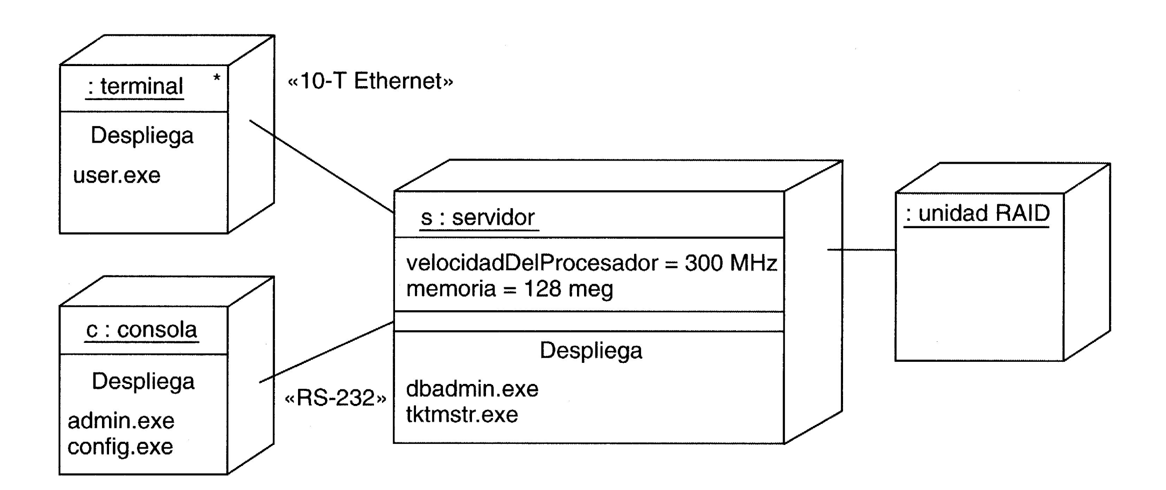
- Los componentes se utilizan para modelar los elementos físicos que pueden hallarse en un nodo, tales como ejecutables, bibliotecas, tablas, archivos y documentos
- Un componente es una parte física y reemplazable de un sistema que conforma con un conjunto de interfaces y proporciona la realización de esas interfaces



Diagramas de despliegue

[Booch 99, capítulo 26]

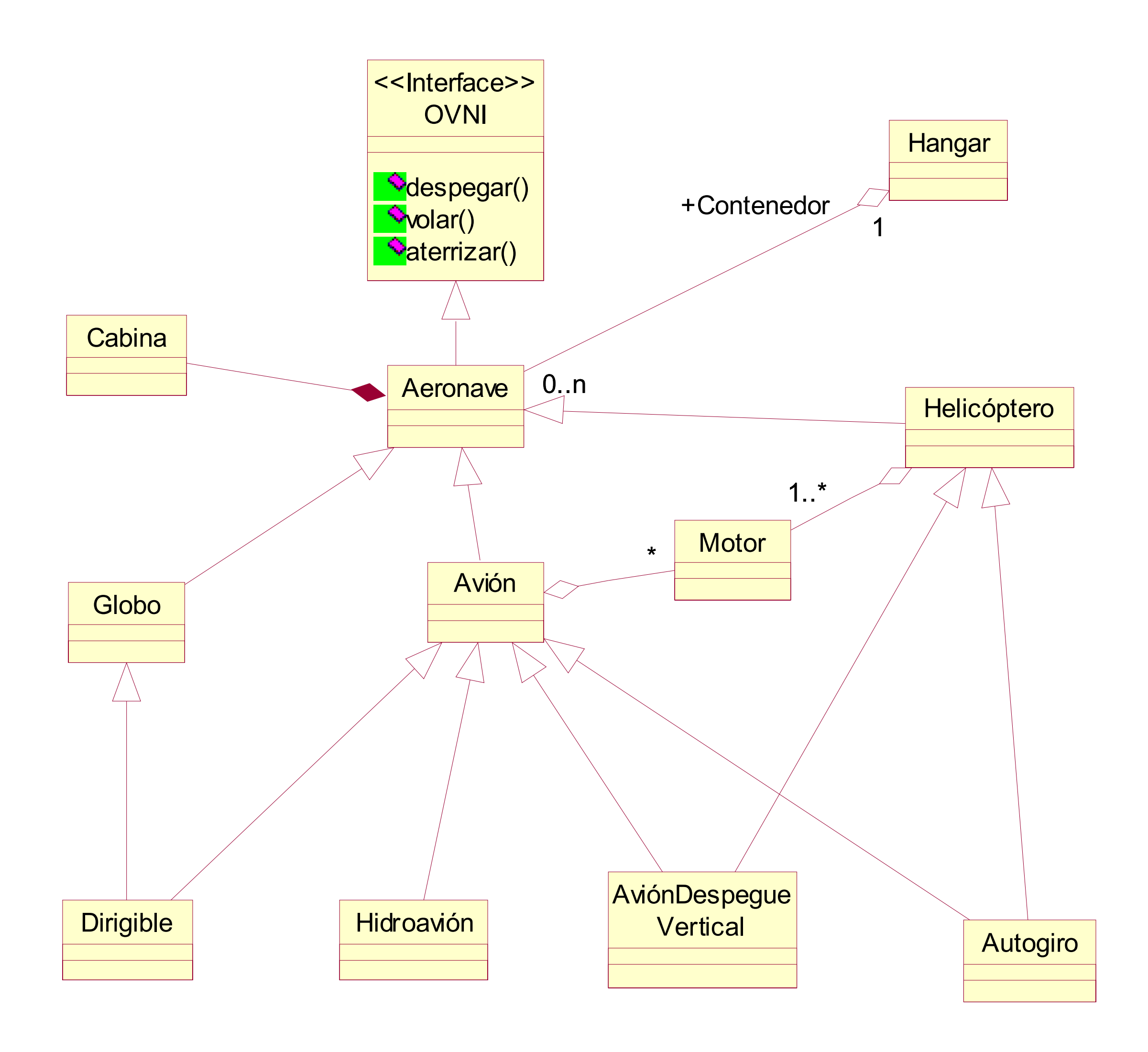
- Representan la topología del hardware
- Un nodo es un elemento físico que existe en tiempo de ejecución y que representa un recurso computacional



Ejercicios de auto-evaluación (I)

En la República Manzanera Vetustosa se quiere construir un aeropuerto en su capital Carbayonia, el software se encarga a la empresa Chapuzosa y debe ser según el pliego de condiciones totalmente Orientado a Objetos. Dado que los técnicos de Chapuzosa no tienen experiencia en Análisis y Diseño Orientado a Objetos, se le pide su opinión.

Se ha desarrollado el siguiente diagrama de clases:



Ejercicios de auto-evaluación (II)

1. Los de Chapuzosa diseñan el diagrama de clases en UML en la figura anterior. Se les puede decir que **A)** La vida (creación y destrucción) de los objetos de la clase Motor depende de la vida de los objetos de las clases Avion y Helicoptero **B)** Los objetos de la clase Motor se pueden crear y se destruir de forma independiente de las clases Avion y Helicoptero **C)** Los objetos de la clase Cabina se pueden crear y se destruir de forma independiente de la clase Aeronave **D)** Todas las respuestas son correctas **E)** Ninguna respuesta es correcta
2. Los de Chapuzosa están experimentando con la herencia, tal y como se observa en el diagrama anterior y te preguntan: ¿Cuántos objetos de la clase Cabina tiene un objeto de la clase Hidroavión? **A)** ninguno **B)** 1 **C)** Depende de la implementación **D)** Todas las respuestas anteriores son correctas **E)** Ninguna respuesta anterior es correcta.
3. Los de Chapuzosa se encuentran el diagrama anterior donde Hangar es un contenedor polimórfico de punteros a objetos Aeronave, y se preguntan ¿Qué cosas puede contener?. **A)** El Hangar puede contener punteros a objetos de la clase Dirigible **B)** El hangar puede contener directamente punteros a objetos Motor **C)** Hangar puede contener punteros a objetos Cabina **D)** Todas las respuestas anteriores son correctas **E)** Ninguna respuesta anterior es cierta
4. Los de Chapuzosa discuten sobre las interfaces de UML, y dicen que **A)** Los objetos Autogiro tienen un método volar **B)** Se pueden implementar en C++ usando clases abstractas con métodos abstractos puros **C)** La clase Aeronave debe implementar el método despegar **D)** Todas las respuestas anteriores son correctas **E)** Ninguna respuesta anterior es correcta
5. Los de Chapuzosa discuten sobre Patrones de Diseño, y dicen que el patrón Singleton es **A)** Un patrón de creación **B)** Es un patrón de comportamiento **C)** Un patrón estructural **D)** Todas las respuestas anteriores son correctas **E)** Ninguna respuesta anterior es correcta.
6. Los de Chapuzosa discuten sobre Actores de UML, y dicen que **A)** Se identifican en el Análisis **B)** Se implementan con clases **C)** Son los que interactúan con el sistema **D)** Todas las respuestas anteriores son correctas **E)** Ninguna respuesta anterior es correcta
7. Los de Chapuzosa discuten sobre prototipos, y dicen que **A)** No son un tipo de diagrama UML **B)** Son para evaluar los requisitos **C)** Son parte del Análisis **D)** Todas las respuestas anteriores son correctas **E)** Ninguna respuesta anterior es correcta
8. Los de Chapuzosa discuten sobre escenarios, y dicen que **A)** Son especializaciones de los casos de uso **B)** Son parte del modelo análisis **C)** No están especificados en UML **D)** Todas las respuestas anteriores son correctas **E)** Ninguna respuesta anterior es correcta
9. Los de Chapuzosa discuten sobre los Diagramas de Interacción, y dicen que **A)** Modelan aspectos dinámicos del sistema **B)** Son parte del diseño **C)** Son un caso particular de los Escenarios **D)** Todas las respuestas anteriores son correctas **E)** Ninguna respuesta anterior es correcta
10. Los de Chapuzosa discuten sobre Diagramas de Objetos, y dicen que **A)** Es necesario hacerlos todos **B)** Son parte del modelo físico **C)** Muestran los objetos y sus relaciones en un instante determinado **D)** Todas las respuestas anteriores son correctas **E)** Ninguna respuesta anterior es correcta

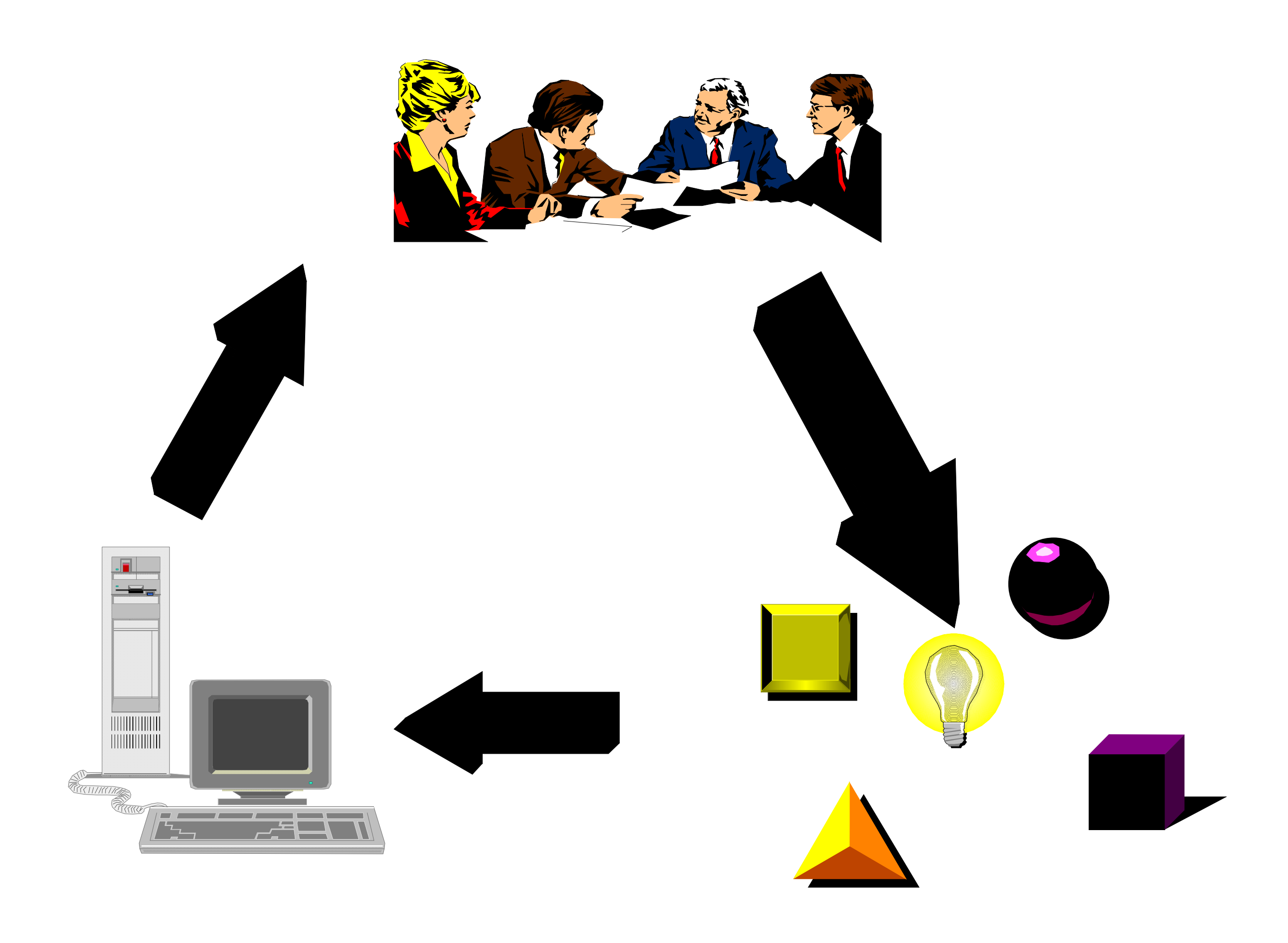
Respuestas: 1-b), 2-b), 3-a),4-d), 5-a), 6-d), 7-d) , 8-d), 9-d), 10-c)

Referencias

- [Booch 94] G.Booch. *Object-oriented analysis and design with applications.* Benjamin Cummings (1994). Versión castellana: *Análisis y diseño orientado a objetos con aplicaciones.* 2ª Edición. Addison-Wesley/ Díaz de Santos (1996).
- [Booch 99] G. Booch, J. Rumbaugh, I. Jacobson. *The unified modeling language user guide.* Addison-Wesley (1999). Versión castellana *El lenguaje unificado de modelado.* Addison-Wesley (1999)
- [Coad 97] P. Coad, M. Mayfield. *Java design*. Prentice-Hall, 1997.
- [Cook 94] S. Cook and J. Daniels, *Designing Object Systems: Object-oriented Modelling with Syntropy,* Prentice-Hall Object-Oriented Series, 1994.
- [Eriksson 98] H-E Eriksson & M. Penker. *UML Toolkit. Wiley, 1998.*
- [Fowler 97] M. Fowler with K. Scott, *UML Distilled: Applying the Standard Object Modeling Language*, ISBN 0-201-32563-2, Addison-Wesely, 1997.
- [Gamma 95] Gamma E. et al. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.Versión en castellano *Patrones de Diseño*. Pearson Educación, 2002.
- [Granham 97] I. Graham, B. Henderson-Sellers, H. Younessi. *The OPEN Process Specification*. Addison-Wesley (1997).
- [Grand 98] Grand M. *Patterns in Java. Volume 1.* Wiley, 1998.
- [Grand 99] Grand M. *Patterns in Java. Volume 2.* Wiley, 1999.
- [Henderson-Sellers, 97] B. Henderson-Sellers. *OML: OPEN Modelling Language*. SIGBOOKS, 1997.
- [Jacobson 92] I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard. *Object-Oriented software Engineering. A use case driven Approach.* Addison-Wesley (1992)
- [Jacobson 99] I. Jacobson, G. Booch, J. Rumbaugh. *The unified software development process*. Addison-Wesley (1999).
- [Larman 98] C. Larman. *Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design*. Prentice-Hall (1998). Versión castellana: *UML y patrones. Introducción al análisis y diseño orientado a objetos*. Prentice-Hall (1999).
- [Lee 97] R. C.Lee & W. M. Tepfenhart. *UML and C++, Prentice-Hall, 1997*
- [López 97] N. López, J. Migueis, E. Pichon. *Intégrer UML dans vos projects.* Editions Eyrolles, 1998. Versión castellana: *Integrar UML en los proyectos. Ediciones Gestión 2000 (1998).*
- [Muller 97] Muller P-A *Modélisation Object avec UML* Eyrolles 1997. Versión castellana: *Modelado de objetos con UML, Gestión-2000, 1997*
- [Odell 98] J.J. Odell. *Advanced Object-Oriented Analysis & Design Using UML*. SIGS, 1998.
- [OMG] *www.omg.org*
- [OPEN] *www.open.org.au*
- [Piattini 96] M.G. Piattini, J.A. Calvo-Manzano, J. Cervera, L. Fernández. *Análisis y diseño detallado de aplicaciones de gestión*. RA-MA (1996)
- [Rational] UML y herramienta Rational Rose en *www.rational.com*
- [Rumbaught 91] Rumbaught J., Blaha M., Premerlani W., Wddy F., Lorensen W. *Object-oriented modeling and design.* Prentice-Hall (1991). Versión castellana: *Modelado y diseño orientado a objetos. Metodología OMT*. Prentice-Hall (1996)
- [Rumbaught 99] Rumbaught J., I. Jacobson, G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley (1999)
- [Reenskaug 96] T. Reenskaug. *Working with Objects. The Ooram Software Engineering Method.*Prentice-Hall, 1996
- [Wilkinson 95] N. M. Wilkinson. *Using CRC Cards. An Informal Approach to Object-Oriented Development*, 1995, SIGS BOOKS, 1-884842-07-0

Tema 7º
El proceso de desarrollo

- Principios básicos
- El microproceso de desarrollo
- El macroproceso de desarrollo
- Resumen



Principios básicos

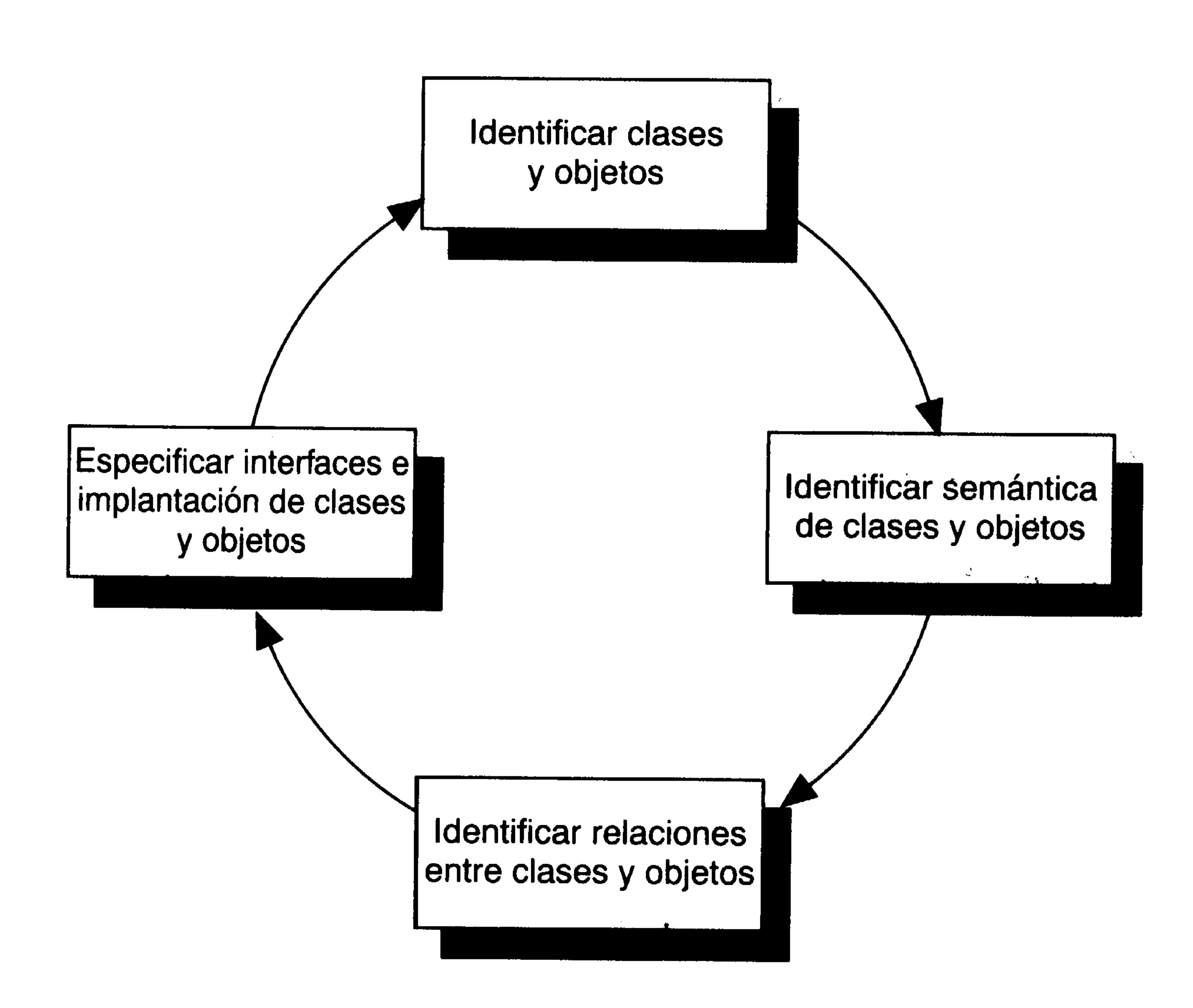
[Booch 94]

- No hay recetas mágicas
- Características fundamentales de un proyecto con éxito
  - **Buena visión arquitectónica**
    - No existe ningún camino bien definido para idear una arquitectura. Tan sólo se pueden definir los atributos de una buena arquitectura:
      - Capas de abstracción bien definidas
      - Clara separación de intereses entre interfaz e implementación
      - Arquitectura simple
    - Es necesario distinguir entre decisiones estratégicas y tácticas
    - ***Decisiones estratégicas*** *es aquella que tiene amplias implicaciones estratégicas e involucra así a la organización de las estructuras de la arquitectura al nivel más alto*
    - ***Decisiones tácticas*** *son las que sólo tienen implicaciones arquitectónicas locales, es decir sólo involucran a los detalles de interfaz e implementación de una clase*
  - **Ciclo de vida incremental e iterativo**
    - Los ciclos de desarrollo no deben ser anárquicos ni excesivamente rígidos
    - Cada pasada por un ciclo análisis/diseño/evolución lleva a refinar gradualmente las decisiones estratégicas y tácticas, convergiendo en última instancia hacia una solución con los requisitos reales de los usuarios finales (habitualmente no expresados explícitamente por éstos)

El microproceso de desarrollo

[Booch 94]

Representa las actividades diarias del desarrollador individual o de un equipo pequeño de desarrolladores



Identificación de clases y objetos

- **Propósito:** obtener las clases y objetos como abstracciones del vocabulario del dominio del problema
- **Producto:** construir el ***diccionario de clases***
  - depósito de información (repository)
    - simple base de datos
    - Herramienta CASE
- **Actividades:** Descubrimiento e invención
  - Diagramas de Casos de Uso
  - Definición de los Escenarios Principales
- **Hitos:** se completa con éxito esta fase cuando se dispone de un diccionario estable
- **Medida de la bondad:** Cuando el diccionario no sufre cambios radicales cuando se itera a través del microproceso

Identificación de la semántica de las clases y objetos

- **Propósito:** Establecer el comportamiento y los atributos de cada abstracción identificada en la fase previa
- **Productos**
  - Se obtienen tres productos
    - **Refinamiento del diccionario de clases**
    - **Diagramas de Secuencia (UML)**
    - **Diagramas de Colaboración (UML)**
  - Se pueden utilizar bases de datos o herramientas
  - La incapacidad para especificar una semántica clara es un signo de que las propias abstracciones son defectuosas
- **Actividades**
  - Se realizan tres actividades
    - Narración de sucesos (*storyboarding*): Se centra en el comportamiento no en la estructura
      - Seleccionar escenario
      - Identificar las abstracciones
      - Relatar la actividad en el escenario
      - Iterar reasignando responsabilidades
    - Diseño de clases aisladas
      - Seleccionar la abstracción y enumerar sus papeles y responsabilidades
      - Idear un conjunto de operaciones suficiente que satisfagan las responsabilidades
      - Considerar las operaciones una a una y asegurarse de que son primitivas
      - Dejar para más tarde las necesidades de construcción, copia y destrucción
      - Considerar la necesidad de completud
    - Uso de patrones de diseño para diseñar la colaboración entre clases
      - En el conjunto completo de escenarios buscar patrones comunes de interacción entre abstracciones
      - En el conjunto completo de responsabilidades buscar patrones de comportamiento
      - En el conjunto de operaciones buscar patrones en la forma en que se realizan las operaciones (debe hacerse en las fases tardías del ciclo de vida)
- **Hitos**: Se completa con éxito cuando para cada abstracción se tiene un conjunto de responsabilidades y/o operaciones razonablemente suficiente, primitivo y completo
- **Medidas de bondad**: Las responsabilidades que no son simples ni claras sugieren que la abstracción dada aún no está bien definida

Identificación entre las relaciones de clases y objetos

- **Propósito**
  - Es consolidar las fronteras y reconocer los colaboradores de cada abstracción que se identificó previamente en el microproceso
- **Productos**
  - Se producen seis:
    - **Diagramas de Clases** (UML). Organizados en subsistemas y módulos.
    - **Diagramas de Colaboración** (UML). Más detallados.
    - **Diagramas de Actividad** (UML)
    - **Diagramas de Estados** (UML)
    - **Diagramas de Componentes** (UML)
    - **Diagramas de Despliegue** (UML)
  - En esta fase no es deseable (ni posible) expresar todos los diagramas que expresen todas las visiones concebibles. *Tan sólo se representan las más interesantes*
- **Actividades**
  - Especificación de asociaciones
    - Tomar un conjunto de clases para un nivel de abstracción
    - Considerar la dependencia dos a dos clases
    - Para cada asociación especificar el papel de cada participante, su cardinalidad y las restricciones
    - Validar las decisiones de diseño recorriendo los escenarios
  - Identificación de varias colaboraciones
    - Ilustrar la dinámica de las asociaciones de la actividad anterior en el diagrama de objetos
    - Buscar jerarquías de generalización/especialización (herencias)
    - Agrupar las clases en categorías y organizar los módulos en subsistemas
    - Capturar las decisiones anteriores en diagramas de módulos
  - Refinamiento de las asociaciones
    - Buscar nuevos patrones en las jerarquías generalización/especialización
    - Si hay patrones de estructura considerar la creación de clases que capturen la nueva estructura
    - Buscar de similar comportamiento. Hermanas en el diagrama de herencias o generalizables mediante clases parametrizadas (Templates)
    - Considerar la posibilidad de simplificar las asociaciones existentes para mejorar su manejo
    - Introducir nuevos detalles tácticos: papeles, claves, cardinalidad, ...
- **Hitos**
  - Se considera finalizada esta fase cuando las abstracciones se han concretado de forma que pueden servir como planos para dirigir la implementación
- **Medidas de bondad**
  - Si las abstracciones son difíciles de implementar será síntoma de que no se ha ideado un conjunto significativo de relaciones entre ellas

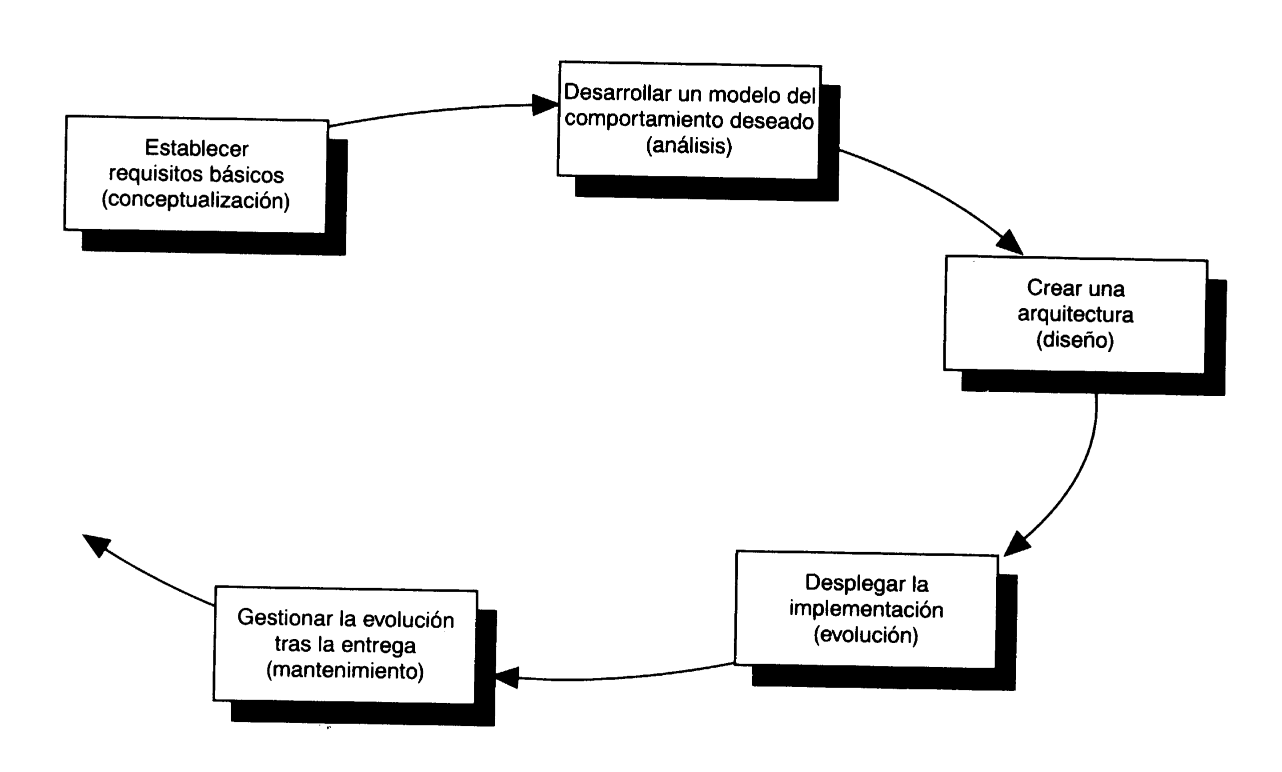
Implementación de clases y objetos

- **Propósito:** Implementar las clases y objetos
- **Productos**
  - Un conjunto de ficheros con pseudocódigo y códigos fuente en algún lenguaje de programación
    - Se pueden generar partes con las herramientas
  - Se actualiza el diccionario de clases incluyendo la nueva información de la implementación
- **Actividades:** Selección de las estructuras de datos y algoritmos que suministran la semántica de las abstracciones identificadas previamente
  - Para cada clase identificar su protocolo con las operaciones
  - Antes de elegir una representación construida totalmente, tratar de usar la genericidad
  - Procurar que los objetos no todas las operaciones y delegar en otros objetos. esto puede obligar a algún reajuste de responsabilidades
  - Tratar de usar estándares antes de crear algo propio
  - Seleccionar el algoritmo óptimo para cada operación
- **Hitos**: Se finaliza cuando se obtiene un modelo ejecutable o módulos ejecutables
- **Medidas de bondad:** Las implementaciones complejas o ineficientes indican que las abstracciones son pobres o tienen carencias

El macroproceso de desarrollo

[Booch 94]

Es el marco de referencia para controlar el microproceso



Conceptualización

Establecer los requisitos principales del software que se va a construir

- **Propósito:** *Establecer los requisitos esenciales para el sistema*
- **Productos:** *los prototipos*
  - Todos los prototipos están destinados a ser desechados
- **Actividades:** *No tiene reglas rígidas*
  - La conceptualización es una actividad creativa
  - Se pueden dar unas vías de exploración
    - Establecer objetivos
    - Reunir un equipo apropiado para hacer el prototipo
    - Evaluar el prototipo y tomar decisiones para exploraciones posteriores
- **Hitos:** *Establecer criterios explícitos para la terminación de prototipos*
- **Medidas de la bondad:** *El prototipo se considera aceptable cuando se puede dar el salto de concepto a producto*

Análisis

Desarrollar un modelo del comportamiento deseado del sistema

- **Propósito:** *Proporcionar una descripción del problema*
  - El análisis se centra en el comportamiento identificando los puntos funcionales
    - Los puntos funcionales denotan los comportamientos de un sistema observables exteriormente y comprobables
  - Para comprobar si el análisis es completo se usa la trazabilidad para comprobar que no se ha olvidado ningún punto funcional
- **Productos**
  - Casos de uso
  - Los escenarios: cada escenario denota algún punto funcional
  - Documento formal de análisis que establece los requisitos del comportamiento del sistema
  - Identificación de áreas de riesgos técnicos
- **Actividades**
  - Análisis del dominio
    - Identificar clases y objetos
  - Planificación del escenario
    - Identificar los puntos funcionales principales del sistema
    - Para cada punto funcional importante realizar una narración de sucesos
    - Generar escenarios secundarios para ilustrar comportamientos en condiciones excepcionales
    - En los ciclos de vida claves de objetos generar la maquina de estados finitos de la clase del objeto
    - Seleccionar patrones de diseño entre los escenarios
    - Actualizar el diccionario de clases
- **Hitos:** *Finaliza cuando se han desarrollado y comprobado los escenarios con todos los comportamientos fundamentales del sistema*
  - La comprobación debe realizarse por los expertos en el dominio, los usuarios y los analistas
- **Medidas de la bondad**
  - Deberá comunicar una visión del completa y simple del sistema a todo el equipo de desarrollo
  - Informará a todo el equipo de desarrollo de las estimaciones de los riesgos

Diseño

Crear una arquitectura para la implementación

- **Propósito***: Crear una arquitectura para la implementación y establecer las políticas tácticas comunes que deben utilizarse por parte de los elementos dispares del sistema*
- **Productos**
  - Descripción de la arquitectura
    - Diagramas (secuencia, colaboración, clases, objetos,…)
    - Definición de Subsistemas. Agrupaciones de clases y módulos
  - Descripciones de políticas tácticas comunes
    - Descripción mediante escenarios. Diagramas de Interacción.
- **Actividades**
  - Planificación arquitectónica: *Proyectar las capas y particiones del sistema*
    - Agrupación de los puntos funcionales del análisis en capas y particiones de la arquitectura
    - Validar la arquitectura creando una versión ejecutable
    - Evaluar los puntos fuertes y débiles de la arquitectura
  - Diseño táctico
    - Enumerar las políticas comunes que deben seguir los elementos dispares de la arquitectura
    - Para cada política común desarrollar un escenario que describa la semántica de esa política
    - Documentar cada política y efectuar recorridos siguiéndola por toda la arquitectura
  - Planificación de versiones: Fijan el teatro donde se debe desenvolver la evolución de la arquitectura
    - Organizar los comportamientos de los escenarios siguiendo un orden (de escenarios fundamentales a secundarios)
    - Asignar puntos funcionales a cada versión
    - Ajustar objetivos y planes a cada versión, permitiendo un tiempo suficiente para documentación, pruebas y sincronización con otras actividades de desarrollo
    - Comenzar a planificar tareas y recursos necesarios para cada versión
- **Hitos**
  - Se valida la arquitectura mediante un prototipo y su revisión
  - Plan para futuras versiones
- **Medidas de la bondad:** *la simplicidad*

Evolución

Transformar la implementación mediante refinamientos sucesivos

- **Propósito:** *Aumentar y cambiar la implementación mediante refinamientos sucesivos*
- **Productos:** *Conjunto de versiones sucesivas y refinadas*
- **Actividades**
  - Aplicación del microproceso
  - Gestión de cambios
- **Hitos:** *cuando la funcionalidad y calidad de las versiones son suficientes para expedir el producto*
- **Medidas de la bondad:** *Muchos cambios en los interfaces arquitectónicos y en las políticas tácticas indican mala calidad de la evolución*

Mantenimiento

Gestionar la evolución postventa o postentrega

- **Propósito:** *Gestionar la evolución postventa*
- **Productos:** *Iguales a los de la fase de evolución*
- **Actividades:** *Son iguales a las de la fase de evolución más las siguientes:*
  - Asignar prioridad a las peticiones de mejoras básicas o informes de errores
  - Considerar que un conjunto amplio de cambios puede ser un nuevo punto funcional
  - Añadir pequeñas mejoras
  - Comenzar a gestionar la siguiente versión
- **Hitos:** *Nueva versión final y versiones intermedias de depuración de errores*
- **Medidas de la bondad***: flexibilidad al cambio*
  - Se entra en la etapa de **conservación** *si la respuesta a las mejoras exige recursos de desarrollo elevados*

Resumen

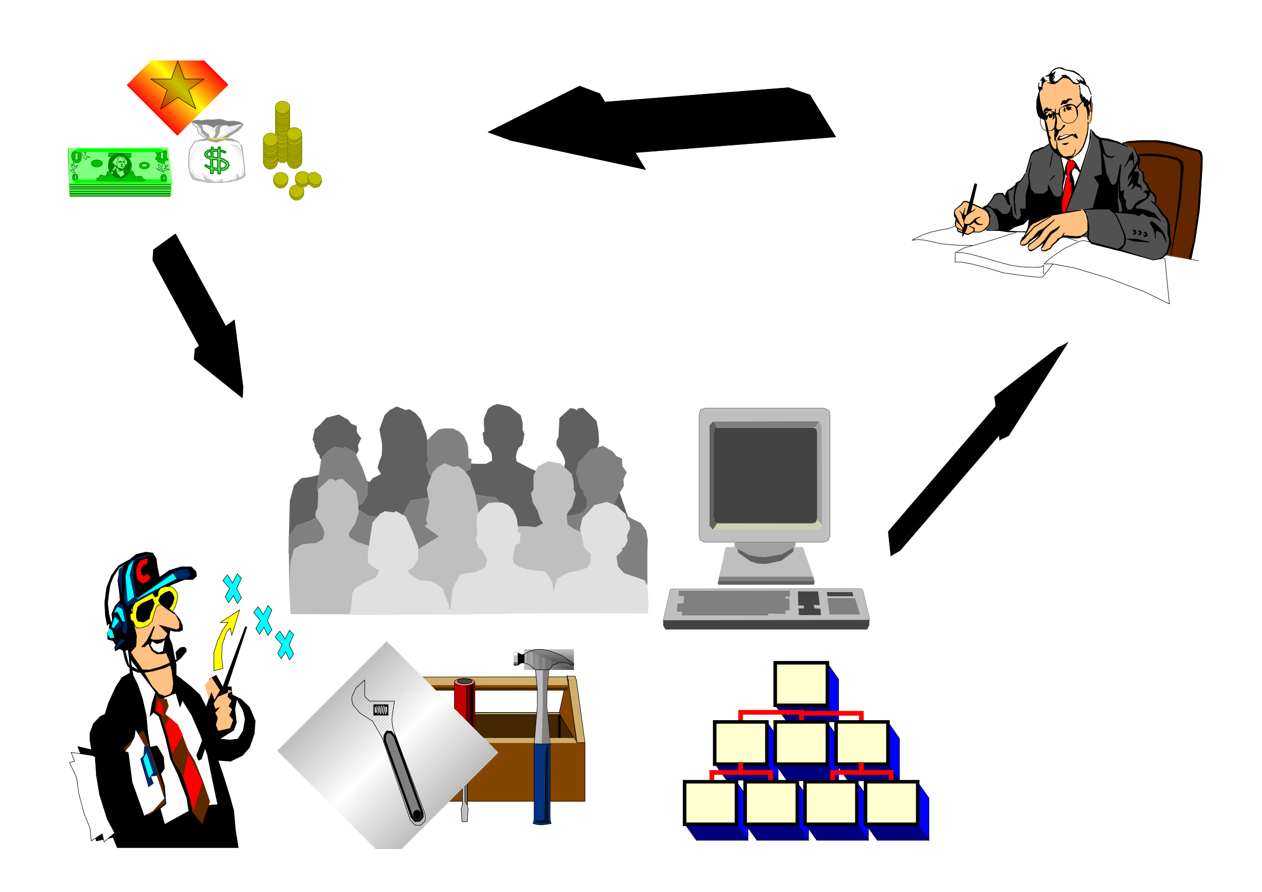
- Los proyectos de desarrollo de software con éxito se caracterizan por una estructura arquitectónica fuerte y por un ciclo de vida bien dirigido, iterativo e incremental
- No hay recetas para el proceso de desarrollo
- El proceso de desarrollo orientado a objetos se puede percibir desde los puntos de vista del microproceso y del macroproceso
- El microproceso representa las actividades diarias del equipo de desarrollo
- El macroproceso es el marco de referencia que controla el microproceso y define una serie de productos medibles y actividades para gestionar el riesgo

Referencias

- **[Booch 1994]** G.Booch. *Object-oriented analysis and design with applications.* Benjamin Cummings (1994). Versión castellana: *Análisis y diseño orientado a objetos con aplicaciones.* 2ª Edición. Addison-Wesley/ Díaz de Santos (1996).
- **[Booch 1999]** G. Booch, J. Rumbaugh, I. Jacobson. *The unified modeling language user guide.* Addison-Wesley (1999). Versión castellana *El lenguaje unificado de modelado.* Addison-Wesley (1999)
- **[Buschman 1996]** F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, M. Stal. *Pattern-Oriented software architecture. Volume 1*. Wiley, 1996.
- **[Buschman 2000]** F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, M. Stal. *Pattern-Oriented software architecture. Volume 2*. Wiley, 2000.
- **[Cook 1994]** S. Cook and J. Daniels, *Designing Object Systems: Object-oriented Modelling with Syntropy*, Prentice-Hall Object-Oriented Series, 1994.
- **[Cooper 2000]** J.W. Cooper. *Java Design Patterns. Addison*-Wiley, 2000.
- **[Gamma 1995]** Gamma E. et al. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- **[Grand 1998]** Grand M. *Patterns in Java. Volume 1*. Wiley, 1998.
- **[Grand 1999]** Grand M. *Patterns in Java. Volume 2*. Wiley, 1999.
- **[Jacobson 92]** I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard. *Object-Oriented software Engineering. A use case driven Approach.* Addison-Wesley (1992)
- **[Jacobson 1999]** I. Jacobson, G. Booch, J. Rumbaugh. *The unified software development process*. Addison-Wesley (1999).Versión Castellana *El Proceso Unificado de Desarrollo de Software, 1999.*
- **[Piattini 1996]** M.G. Piattini, J.A. Calvo-Manzano, J. Cervera, L. Fernández. *Análisis y diseño detallado de aplicaciones de gestión*. RA-MA (1996)
- **[Rumbaught 1991]** Rumbaught J., Blaha M., Premerlani W., Wddy F., Lorensen W. *Object-oriented modeling and design*. Prentice-Hall (1991). Versión castellana: *Modelado y diseño orientado a objetos. Metodología OMT*. Prentice-Hall (1996)
- **[Rumbaught 1999]** Rumbaught J., I. Jacobson, G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley (1999). Versión Castellana *El lenguaje Unificado de Modelado. Manual de Referencia.* Addison-Wesley, 1999.

Tema 8º: Aspectos prácticos

- Gestión y planificación
- Administración de personal
- Gestión de versiones
- Reutilización
- Control de calidad del software
- Documentación
- Herramientas
- Temas especiales
- Las ventajas y los riesgos del desarrollo orientado a objetos



Gestión y planificación

Es necesario un liderazgo fuerte que gestione y dirija el proyecto de desarrollo de software

- Gestión del riesgo
  - Riesgos técnicos: *selección de componentes, elementos genéricos, tramas de herencias, herramientas CASE,…*
  - Riesgos no técnicos: *fechas de entrega, presupuestos, relaciones con el cliente, relaciones entre los miembros del equipo,...*
- Planificación de tareas
  - Frecuencias mínimas de reuniones para favorecer la comunicaciones. Demasiadas reuniones destruyen la productividad
  - Planificar las entregas del macro-proceso
  - Calibración de tiempos del equipo de desarrollo
- Recorridos de inspección
  - La dirección del proyecto debe revisar los aspectos del sistema que conlleven decisiones de desarrollo estratégicas
  - En el análisis pueden hacer revisiones incluso personal no informático para validar los escenarios planteados

Administración de personal

- Asignación de recursos
  - El primer proyecto orientado a objetos necesitará más recursos debido a la curva de aprendizaje inherente a la adopción de una nueva tecnología
  - Para el análisis los requisitos no cambian mucho respecto a otras metodologías no orientadas a objetos
- Papeles en el equipo de desarrollo
  - Papeles principales
    - Arquitecto del proyecto
    - Jefe de subsistema
    - Programador de aplicación
  - Otros papeles
    - Analista
    - Ingeniero de reutilización
    - Responsable de control de calidad
    - Jefe de integración de subsistemas
    - Documentalista
    - Responsable de herramientas
    - Administrador del sistema



Gestión de versiones

- Integración: múltiple prototipos y versiones de producción
  - Versiones internas (pruebas alfa)
  - Versiones de prueba externas (pruebas beta)
  - Versiones definitivas
- Gestión de configuraciones y control de versiones
  - El código fuente no es el único producto del desarrollo que debería ponerse bajo la gestión de configuraciones y control de versiones. Los mismos conceptos se aplican a los productos:
    - Requisitos
    - Diagramas (interacción, clases, objetos, módulos, procesos)
    - Código fuente
    - Documentación
- Prueba
  - Prueba unitaria de clases y mecanismos
  - Prueba de subsistema. Se siguen los escenarios del subsistema
  - Prueba de sistema. Se siguen los escenarios

Reutilización

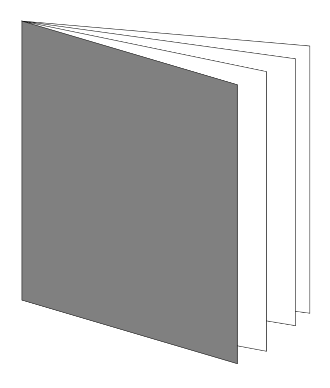
- Elementos de la reutilización
  - Cualquier artefacto del desarrollo de software puede reutilizarse
    - Código (componentes, genericidad, herencia)
    - Diseños (patrones de diseño)
    - escenarios
    - documentación
- Institucionalizar la reutilización
  - Las oportunidades para reutilizar código deben buscarse y recompensarse
  - La reutilización efectiva se alcanza mejor responsabilizando a personas concretas de esa actividad

Control de calidad y métricas

- Calidad del software: *la aptitud de uso del producto completo de software*
  - El uso de tecnologías orientadas a objetos no implica automáticamente software de calidad
  - Una medida de la calidad puede ser la densidad de defectos
    - por cada mil líneas de código
    - por cada subsistema, módulo, o por clase
  - La búsqueda de errores debe hacerse en las pruebas alfa y beta.
- Métricas orientadas a objetos
  - ¿Cómo medir el progreso de la producción de software?
    - Líneas de código fuente (no es apropiado)
    - Horas de trabajo
    - Métrica ciclomática por clase: mide la complejidad de cada clase
    - Porcentaje finalizado de los subsistemas y módulos
    - Métricas especiales para sistemas orientados a objetos
      - Número de métodos por clase
      - Profundidad del árbol de herencias
      - Número de hijos por árbol
      - Acoplamiento entre objetos (conexión entre objetos)

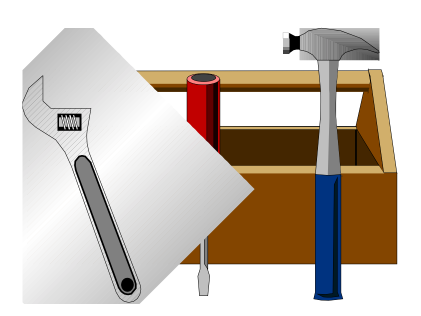
Documentación

- **El legado del desarrollo**: El propio desarrollo también deja una documentación que refleja:
  - Conocimiento sobre el progreso del proyecto
  - Legado de decisiones de análisis y diseño para posibles mantenedores del sistema
- **Contenidos de la documentación**
  - Documentación de la arquitectura de alto nivel del sistema
  - Documentación de las abstracciones y mecanismos clave de la arquitectura
  - Documentación de los escenarios que ilustran el comportamiento práctico de aspectos clave del sistema
- No interesa documentación que describa clase por clase sin explicar sus relaciones



Herramientas

- Tipos de herramientas
  - Elegir herramientas que soporten el cambio de escala
  - Siete tipos de herramientas
    - Herramienta gráfica que soporte la notación
    - Browser (inspector de objetos u hojeador)
    - Compiladores incrementales
    - Depuradores que reconozcan la semántica de clases y objetos
    - Analizadores de rendimiento (profilers)
    - Herramientas de gestión de configuraciones y control de versiones
    - Bibliotecario de clases
    - Constructor de interfaces de usuario
- Implicaciones en la organización
  - Debe haber
    - un responsable de herramientas
    - un ingeniero de reutilización (mantiene la biblioteca de clases)



Temas especiales

- Cuestiones específicas del dominio
  - Diseño de los interfaces de usuario
  - Manejo de bases de datos. Se pueden encapsular como utilidades de clase.
  - Manejo de sistemas en tiempo real
  - Sistemas legados *son aplicaciones para las que existe una gran inversión de capital en software que no pueden abandonarse por razones de economía o seguridad*. Sin embargo sus costes de mantenimiento pueden ser intolerables. Se pueden encapsular como si fueran utilidades de clase.
- Transferencia tecnológica
  - ¿Cómo formar a los desarrolladores orientados a objetos?
  - ¿Qué pendiente tiene la curva de aprendizaje?

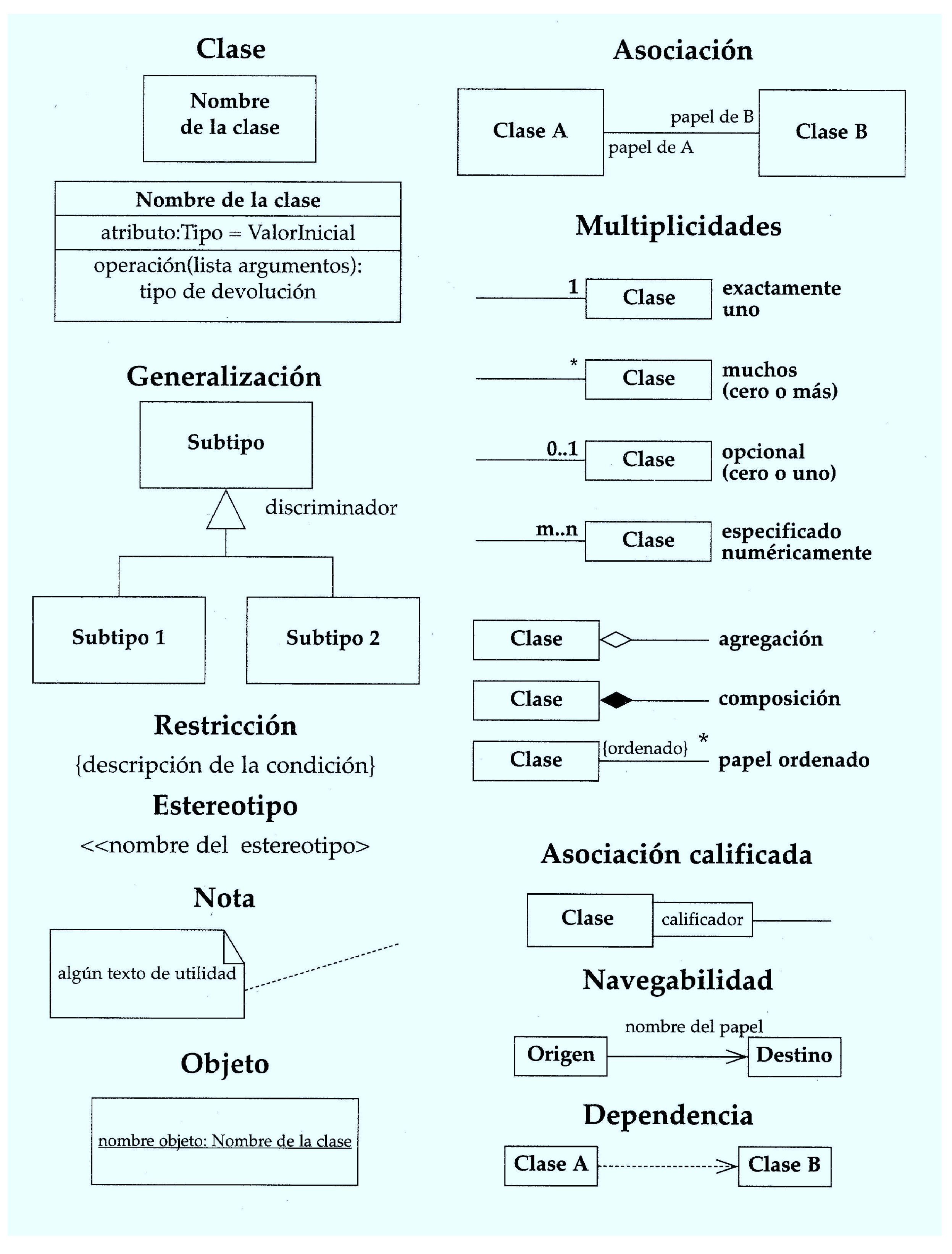
Las ventajas y los riesgos del desarrollo orientado a objetos

- Las ventajas del desarrollo orientado a objetos
  - Competitividad
  - Mayor flexibilidad (reutilización y uso de componentes)
  - Calidad
- Los riesgos del desarrollo orientado a objetos
  - Eficacia
  - Costes de puesta en marcha de una nueva tecnología

Referencias

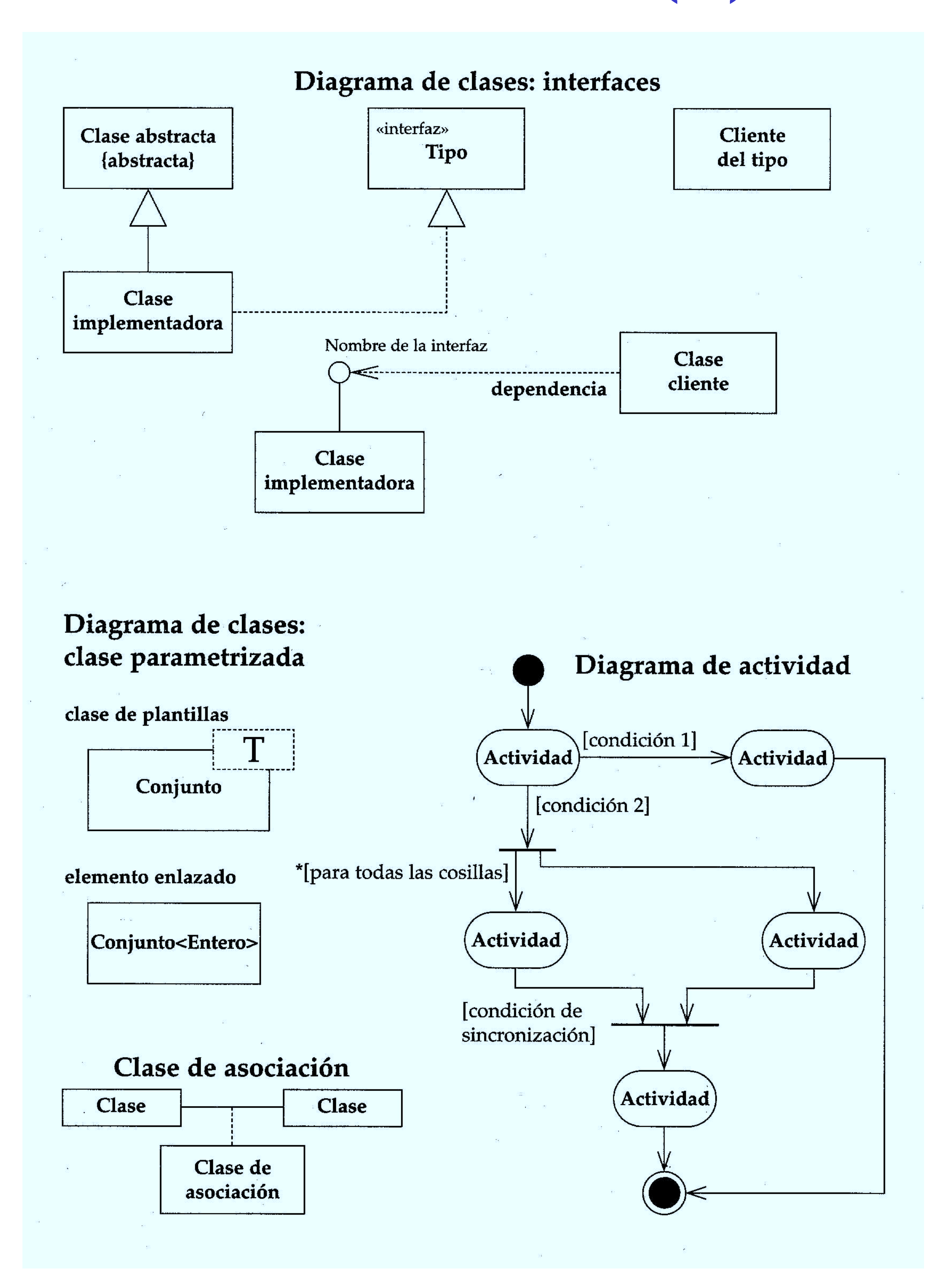
- [Booch 94] G.Booch. *Object-oriented analysis and design with applications.* Benjamin Cummings (1994). Versión castellana: *Análisis y diseño orientado a objetos con aplicaciones.* 2ª Edición. Addison-Wesley/ Díaz de Santos (1996).
- [Booch 99] G. Booch, J. Rumbaugh, I. Jacobson. *The unified modeling language user guide.* Addison-Wesley (1999). Versión castellana *El lenguaje unificado de modelado.* Addison-Wesley (1999)
- [Cook 94] S. Cook and J. Daniels, *Designing Object Systems: Object-oriented Modelling with Syntropy,* Prentice-Hall Object-Oriented Series, 1994.
- [Jacobson 92] I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard. *Object-Oriented software Engineering. A use case driven Approach.* Addison-Wesley (1992)
- [Jacobson 99] I. Jacobson, G. Booch, J. Rumbaugh. *The unified software development process*. Addison-Wesley (1999).
- [Piattini 96] M.G. Piattini, J.A. Calvo-Manzano, J. Cervera, L. Fernández. *Análisis y diseño detallado de aplicaciones de gestión*. RA-MA (1996)
- [Rumbaught 91] Rumbaught J., Blaha M., Premerlani W., Wddy F., Lorensen W. *Object-oriented modeling and design.* Prentice-Hall (1991). Versión castellana: *Modelado y diseño orientado a objetos. Metodología OMT.* Prentice-Hall (1996)
- [Rumbaught 99] Rumbaught J., I. Jacobson, G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley (1999)

ANEXO A: UML (I)

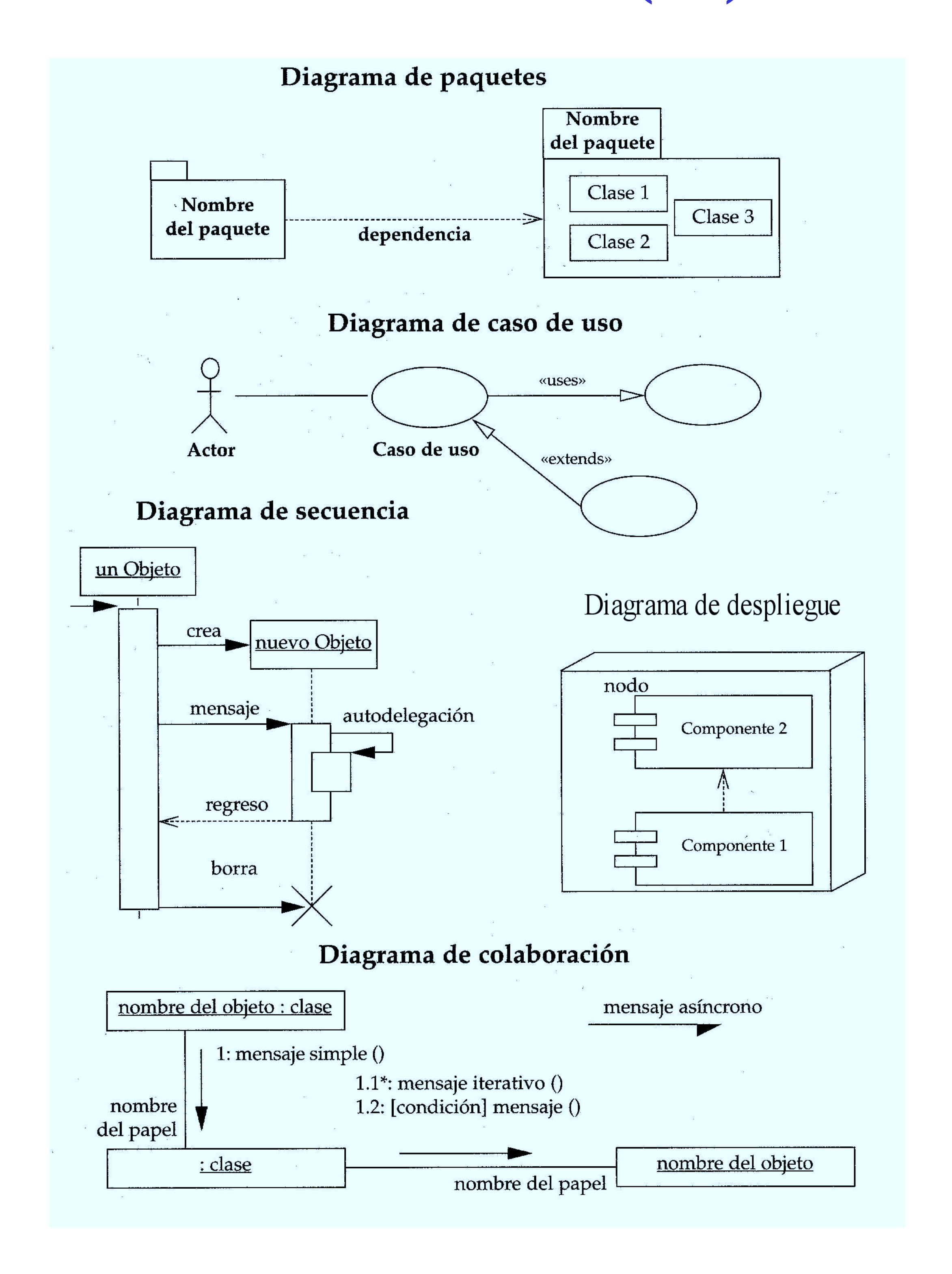


Clase
Nombre
de la clase
Nombre de la clase
atributo:Tipo = ValorInicial
operación(lista argumentos):
tipo de devolución
Generalización
Subtipo
discriminador
Subtipo 1
Subtipo 2
Restricción
{descripción de la condición}
Estereotipo
<<nombre del estereotipo>
Nota
algún texto de utilidad
Objeto
nombre objeto: Nombre de la clase
Asociación
Clase A
papel de B
papel de A
Clase B
Multiplicidades
1
Clase
exactamente
uno
*
Clase
muchos
(cero o más)
0..1
Clase
opcional
(cero o uno)
m..n
Clase
especificado
numéricamente
Clase
agregación
Clase
composición
Clase
{ordenado}
*
papel ordenado
Asociación calificada
Clase
calificador
Navegabilidad
nombre del papel
Origen
Destino
Dependencia
Clase A
Clase B

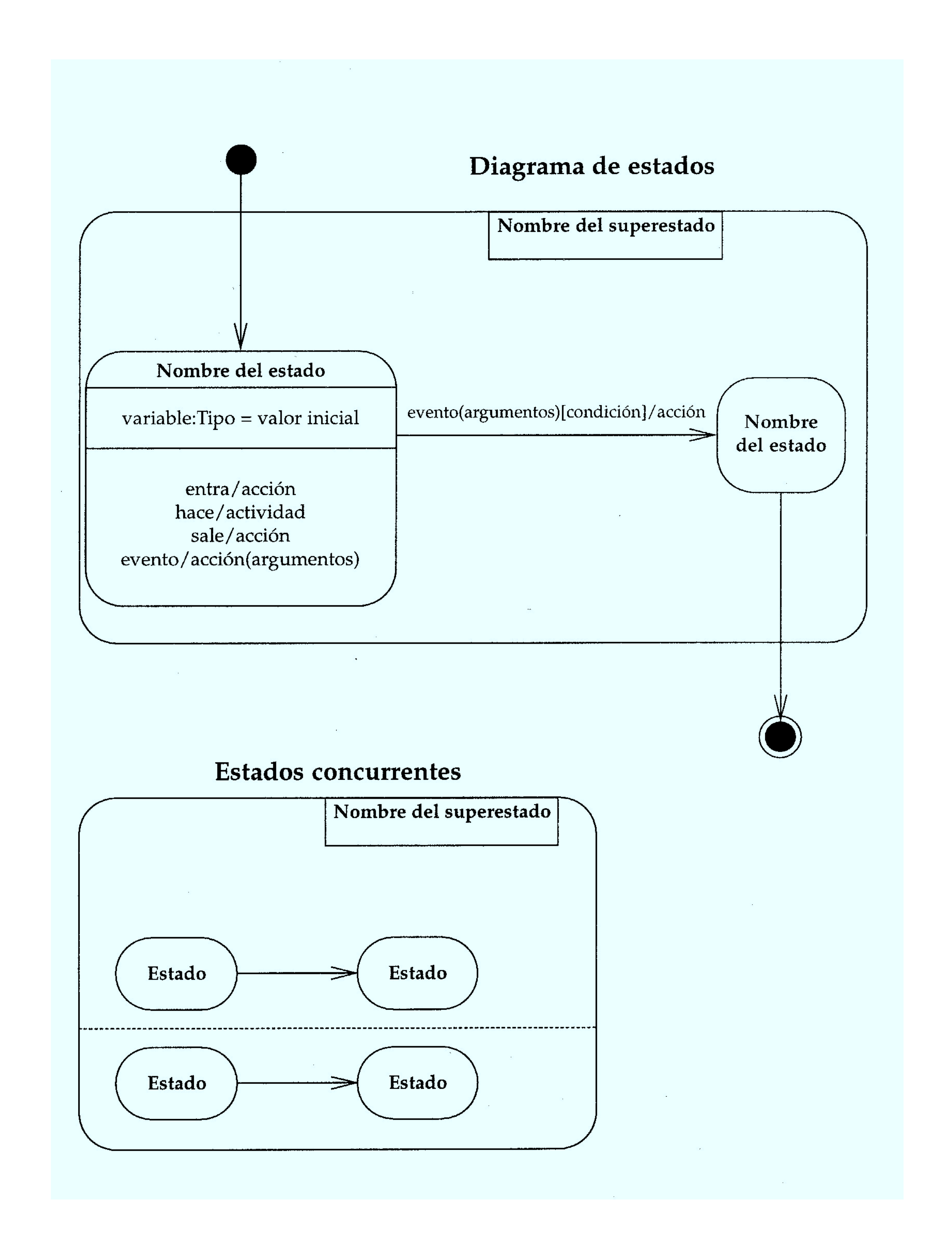
ANEXO A: UML (II)



ANEXO A: UML (III)



ANEXO A: UML (IV)



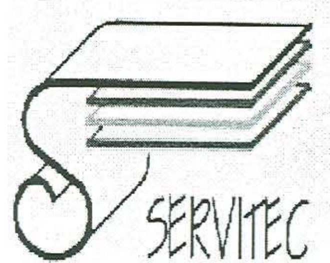


Títulos de la Colección

Ingeniería Informática

Nº CUADERNO	TITULO	ISBN
1	ANÁLISIS Y DISEÑO ORIENTADO A OBJETOS	84-8497-802-8
2	DISEÑO DE PÁGINAS WEB USANDO HTML	84-8497-803-6
3	FORMATOS GRÁFICOS	84-8497-804-4
4	PROGRAMACIÓN EN LENGUAJE C	84-8497-805-2
5	ADMINISTRACIÓN DE WINDOWS N 4.0	84-8497-899-0
6	GUÍA DE USUARIO DE DERIVE PARA WUIDOWS	84-8497-919-9
7	GUÍA DE REFERÉNCIA DE MULTIBASE COSMOS	84-8416-001-7
8	GESTIÓN DE UN TALLER MULTIBASE CÓSMOS	84-8416-034-3
9	EJERCICIOS DE LÓGICA INFOMÁTICA	84-8416-357-1
10	CONEPTOS BÁSICOS DE PROCESADORES DE LENGUAJE	54-8416-889-1
11	INTRODUCCIÓN A LA ADMINISTRACIÓN DE UNIX	84-8416-570-1
12	LÓGICA PROPOSICIONAL PARA LA INFOMÁTICA	84-8416-613-9
13	PROGRAMACIÓN PRÁCTICA EN PROLOG	84-8416-612-0
14	LA HERRAMIENTA CASE META COSMOS	84-699-0054-4
15	GUIA DE LENGUAJE COOL DE MULTIBASE COSMOS	84-699-0053-6
16	GUÍA DE REFERÉNCIA PROGRESS	84-699-2083-9
17	GESTIÓN DE DEPOSITO DENTAL CON PROGRESS	84-699-2082-0
18	GUIA DE DISEÑO Y CONS. REPOSICIÓN MUL. COSMOS	84-699-2081-2
19	GESTIÓN Y ADMIN. DE UN COLEGIO MAYOR MUL-COSMOS	84-699-2080-4
20	MODELADO DE SOFWARE EN UML	84-699-2079-0
21	EL LENGUAJE DE PROGRAMACIÓN JAVA	84-699-3880-0
22	PRINCIPIOS DE ALGORITMIA	84-699-6537-9
23	LISTAS, PILAS Y COLAS	84-699-6536-0
24	RECURSIVIDAD	84-699-6535-2
25	ARBOLES	84-699-6849-1
26	CONJUNTOS Y TABLAS DE DISPERSIÓN	84-699-7398-3
27	ALGORITMOS DE ORDENACIÓN	84-699-7397-5
28	TEORÍA DE GRAFOS	84-699-8362-8
29	IINTR. AL PROCES. EFEC. DE TEXTOS CON MICROSOFT WORD	84-699-9618-5
30	ORACLE SQL	84-688-0420-7
31	TÉCNICAS DE DISEÑO DE ALGORITMOS	84-688-1764-3

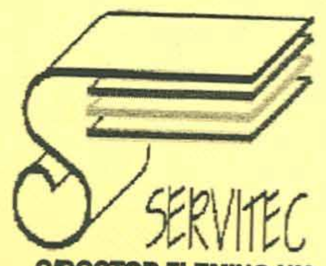
IMPRIME Y DISTRIBUYE



Consultor Editorial
Juan Manuel Cueva Lovelle
cueva@lsi.uniovi.es

C/DOCTOR FLEMING Nº3
33005 OVIEDO
TLF-FAX 985 250581
www.fade.es/copisteriaservitec
E-mail:copisteriaservitec@fade.es

IMPRIME Y DISTRIBUYE



C/DOCTOR FLEMING Nº3
33005 OVIEDO
TLF-FAX 985 250581
www.fade.es/copisteriaservitec
E-mail:copisteriaservitec@fade.es